PhD-FSTC-2015-1
The Faculty of Sciences, Technology and Communication

DISSERTATION

Defense held on 09/01/2015 in Luxembourg

to obtain the degree of

DOCTEUR DE L'UNIVERSITÉ DE LUXEMBOURG
EN INFORMATIQUE

by

Jakub Muszyński
Born on 27 July 1986 in Łapy, Poland

Cheating-Tolerance of Parallel and Distributed Evolutionary Algorithms in Desktop Grids and Volunteer Computing Systems

Dissertation defense committee

Prof. Dr. Pascal Bouvry, dissertation supervisor
Professor, University of Luxembourg

Dr. Francisco Fernández de Vega
Associate Professor, University of Extremadura in Mérida, Spain

Prof. Dr. Ivan Nourdin, Chairman
Professor, University of Luxembourg

Prof. Dr. Franciszek Seredyński, Vice Chairman
Professor, Cardinal Stefan Wyszynski University in Warsaw, Poland

Dr. Sébastien Varrette, dissertation advisor
Research Associate, University of Luxembourg

Abstract

This thesis analyses the fault-tolerant nature of Evolutionary Algorithms (EAs) executed in a distributed environment which is subjected to malicious acts. Such actions are a common problem in Desktop Grids and Volunteer Computing Systems (DGVCS's) utilising idle resources shared by volunteers. Due to the vast computational and storage capabilities provided at a low cost, many large-scale research projects are carried out using such set-ups. However, this advantage is obtained at the expense of a challenging, error prone, heterogeneous and volatile environment of execution.

In the volunteer-based systems, such as BOINC, the incentives offered to the contributors attract also malicious users, commonly called **cheaters**. A cheater typically seeks to obtain the rewards with little or no contribution at all. Additionally, this group may also include "crackers" or "black hat hackers" — users motivated by nothing more than a pure satisfaction from violating computer security.

In this study we use and formalise **cheating faults** — a model for the behaviours described above, which are a subtype of byzantine (arbitrary) faults. They are mainly characterised by the alteration of outputs produced by some or all tasks forming a distributed execution. The approach differs from the arbitrary faults in its implementation, as usually they are introduced intentionally and from within the boundaries of a system.

The innate fault resilience of EAs has been previously observed in the literature. However, this PhD manuscript offers the first, formal analysis of the impact of cheating faults in this area of optimisation techniques. In particular, the following contributions are proposed:

- An in-depth formal analysis of the **cheating-tolerance of parallel Evolutionary Algorithms (EAs)**, including proofs of convergence or non-convergence towards valid solutions in the presence of malicious acts. A step-wise approach is used, focusing firstly on the most simple variant of an EA that is still of theoretical and practical interest, i.e. a (1 + 1) EA. Then the results are extended to regular (population-based) EAs. The analysis shows that the selection mechanism is crucial to achieve convergence of EAs executed in malicious environments.
- The extension of the study to **cheating-resilience of spatially-structured Evolutionary Algorithms (EAs) and gossip protocols**. More precisely, we analyse Evolvable Agent Model (EvAg) relying on Newscast protocol to define neighbourhoods in the evolution and the communication layers. There, we provide the necessary conditions for convergence of the algorithm in a hostile environment and we show that the evolutionary process may be affected only by tampering with the connectivity between the computing resources. After that, we design an effective connectivity-splitting attack which is able to defeat the protocol using very few naïve cheaters. Finally, we provide a set of countermeasures which ultimately lead to a more robust solution.

These results have been published in several international, peer-reviewed venues and well recognized international journals.

By the variety of problems addressed by EAs, this study will hopefully promote their usage in the future developments around distributed computing platforms such as Desktop Grids and Volunteer Computing Systems or Cloud systems where the resources cannot be fully trusted.

Dedicated to my parents

Acknowledgements

I would like to thank my doctoral advisers: Professor Pascal Bouvry, Professor Franciszek Seredyński and Doctor Sébastien Varrette for giving me the opportunity to work on my Ph.D. at the University of Luxembourg. I am grateful for their help and patience during these 4 years of studies. With their support I was able to research what I was interested in, while being well supervised and advised.

I was very happy to work in the team of Professor Bouvry and I thank all its members (former and current) for their collaboration, great work environment and more importantly: for their friendship and all the good moments we spent together.

I am grateful to my family for being supportive in every moment, especially during the writing period.

And finally, I would like to thank my thesis defence committee: Pascal Bouvry, Francisco Fernández de Vega, Ivan Nourdin, Franciszek Seredyński and Sébastien Varrette for their presence and their reviews of this dissertation.

Contents

Chapter 1

Introduction

Contents

This Chapter introduces the thesis context, develops the motivations for the work and presents the different contributions.

1.1 Context

Desktop Grids and Volunteer Computing Systems (DGVCS's) (also known as Global Computing (GC) platforms) are the largest and most powerful distributed computing systems in the world since their advent in the late 1990s. They offer an abundance of computing power at a fraction of the cost of dedicated supercomputers. Such systems, whose typical topology is illustrated in Figure 1.1, are based on volunteer computing: idle cycles of desktop PCs and workstations voluntarily shared by the users worldwide are stolen through the Internet to compute parts of a huge problem. Two of the currently best known projects that exploit this kind of platforms are SETI@home [9] and Folding@home [56], based on Berkeley Open Infrastructure for Network Computing (BOINC) [7]. The first one is aimed to answer the question if we are alone in the universe and the second one aids disease research that simulates protein folding, computational drug design and other types of molecular dynamics).

The latest statistics of BOINC[1] are outstanding: 3.26 millions of users, 12.4 millions of machines and an average computing speed of 120.1 PetaFLOPS.

Many applications from a wide range of scientific domains — including computational biology, climate prediction, particle physics and astronomy — have used the computing power offered by such systems. In general, scheduling is handled in the master/worker model: a server distributes tasks to the participating clients or **workers**; they subsequently process their input data and send the computed results back to the server.

One important aspect of GC platforms is the heterogeneity and the extreme volatility of the resources as their owners may reclaim them without warning, leading therefore to what is commonly named a **crash fault**. In addition, the motivation for the users to contribute are manifold, including the altruistic desire to help or, more importantly, the assignment of credit points proportionally to a user's contribution. These points allow to reward hard-working

[1] See BOINC stats, `http://boincstats.com/`

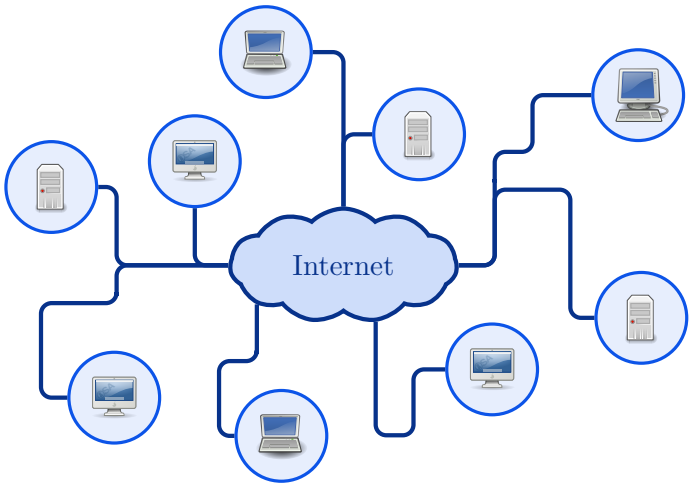


Figure 1.1: A typical Global Computing (GC) platform.

clients in different ways [94]. Unfortunately, incentives also attract cheaters who seek to obtain these rewards with little or no contribution to the system. This is commonly referred as **cheating faults**. Such selfish behavior can be achieved by modifying the client software as experienced in SETI@home [108]. The presence of cheaters in grid computing system is well-known and many countermeasures have been proposed in the literature [44,58,140,163,174], yet at the price of a relatively huge overhead as the only generic approach to detect wrong results created by lazy participants or malicious cheaters relies on duplication, either total or partial. Recently, efficient and generic result checking approaches based on partial task duplication and macro dataflow analysis have been proposed to tackle the issue of massive attacks in which the number of corrupted results exceed a given threshold [163]. Modifications (i.e. cheatings) up to the threshold are assumed to be handled by the application itself, typically by specific low-overhead system-level approaches, such as Algorithm-Based Fault Tolerance (ABFT) techniques where the fault tolerance scheme is tailored to the algorithm performed.

This thesis analyses the ABFT nature of Evolutionary Algorithms (EAs) by evaluating the impact of cheating on computations executed in a distributed and potentially hostile environment such as a GC platform. Evolutionary Algorithms (EAs) are a class of stochastic search techniques which have been successfully applied to solve a large variety of hard optimization problems which typically require several days or months of computation. Decreasing the makespan of the execution is typically obtained by relying on a parallel version executed in a distributed platform. Yet, as mentioned above, such large scale systems are prone to errors.

In this context, this work is an attempt to better understand and characterize the Algorithm-Based Fault Tolerance (ABFT) nature of parallel and distributed EAs against cheating faults, especially from a formal point of view.

1.2 Motivations

Many previous studies in the literature [37,64,75,111,166] suggested through experiments the innate resilience of EAs against crash faults. Also, preliminary results have been obtained against cheating faults [168]. Nevertheless, the following questions remained opened prior to this work:

- Can we formally analyse in which conditions an EA is expected to converge (or not) towards valid solutions despite the presence of cheating faults?
- Which models for EA executions aid the resilience against cheaters?
- And finally, which properties of the models contribute to the innate resilience?

Also, one important aspect of this thesis was to always validate the theoretical results with concrete implementations and experiments. In practice, the High Performance Computing (HPC) platform of the University of Luxembourg (UL) was used for this purpose. To ensure the simulation of large-scale platforms, tremendous computing power were required: more than 768520 CPU Hours[2] was necessary to conduct the experiments proposed in this manuscript. This represents **a total computing effort** of around **87 years and 266 days** over a single GFlop machine... **performed in a 3 years period**.

1.3 Contributions

This section details the contributions as well as the different publications produced during the thesis.
This work covers the following aspects:

- An in-depth formal analysis of the **cheating-tolerance of parallel Evolutionary Algorithms (EAs)**, including proofs of convergence or non-convergence towards valid solutions in the presence of malicious acts. A step-wise approach is used, focusing firstly on the most simple variant of an EA that is still of theoretical and practical interest, i.e. a (1 + 1) EA. Then the results are extended to regular (population-based) EAs. The analysis shows that the selection mechanism is crucial to achieve convergence of EAs executed in malicious environments.
- The extension of the study to **cheating-resilience of spatially-structured Evolutionary Algorithms (EAs) and gossip protocols**. More precisely, we analyse Evolvable Agent Model (EvAg) relying on Newscast protocol to define neighbourhoods in the evolution and the communication layers. There, we provide the necessary conditions for convergence of the algorithm in a hostile environment and we show that the evolutionary process may be affected only by tampering with the connectivity between the computing resources. After that, we design an effective connectivity-splitting attack which is able to defeat the protocol using very few naïve cheaters. Finally, we provide a set of countermeasures which ultimately lead to a more robust solution.

Publications

Whether directly linked to the work presented in this manuscript or through a more general context, this dissertation led to peer-reviewed publications in journals, conference proceedings or books as follows:

- 1 journal article [114];
- 4 articles in international conferences with proceedings and reviews [112, 115, 116, 165];
- 3 articles in international workshops with reviews [113, 117, 164];

It is worth to mention that one of these publications received **the best student paper award** in the reference international conference on Network and System Security (NSS 2014).

[2]The term CPU Hours (processor hours) is a measure of the work done; a CPU Hour is the same as a G-hour, i.e. the measure of the computing work done by one GFlop machine in an hour.

These publications are now detailed.

Peer-Reviewed Journal (1)

[114] J. Muszyński, S. Varrette, P. Bouvry, F. Seredyński, and S. U. Khan. Convergence Analysis of Evolutionary Algorithms in the Presence of Crash-Faults and Cheaters. *Int. Journal of Computers and Mathematics with Applications (CAMWA)*, 64(12):3805–3819, Dec 2012.

International Conferences with proceedings and reviews (4)

[116] J. Muszyński, S. Varrette, J. L. Jiménez Laredo, and P. Bouvry. Exploiting the Hardwired Vulnerabilities of Newscast via Connectivity-splitting Attack. In *Proc. of the IEEE Int. Conf. on Network and System Security (NSS 2014)*, volume 8792 of *LNCS*, pages 152–165, Xian, China, Oct 2014. Springer Verlag.
Best Student Paper Award.

[115] J. Muszyński, S. Varrette, J. L. Jiménez Laredo, and P. Bouvry. Analysis of the Data Flow in the Newscast Protocol for Possible Vulnerabilities. In *Proc. of Int. Conf. on Cryptography and Security System (CSS 2014)*, volume 448 of Communications in Computer and Information Sciences (CCIS), pages 89–99, Lublin, Poland, Sept 2014. Springer.

[112] J. Muszyński, S. Varrette, and P. Bouvry. Expected Running Time of Parallel Evolutionary Algorithms on Unimodal Pseudo-Boolean Functions over Small-World Networks. In *Proc. of the IEEE Congress on Evolutionary Computation (CEC 2013)*, pages 2588–2594, Cancún, Mexico, June 2013. IEEE.

[165] S. Varrette, J. Muszyński, and P. Bouvry. Hash function generation by means of Gene Expression Programming. In *Proc. of Int. Conf. on Cryptography and Security System (CSS 2012)*, Kazimierz Dolny, Poland, Sept 2012. Annales UMCS ser. Informatica.

International Workshops with reviews (4)

[113] J. Muszyński, S. Varrette, and P. Bouvry. On the Resilience of the Newscast Protocol in the Presence of Cheaters. In *2014 Grande Region Security and Reliability Day (GRSRD 2014)*, Saarbrücken, Germany, Mar. 2014.

[117] J. Muszyński, S. Varrette, J. L. Jiménez Laredo, B. Dorronsoro, and P. Bouvry. Convergence of Distributed Cellular Evolutionary Algorithms in Presence of Crash Faults and Cheaters. In *Proc. of the Int. Conf. on Metaheuristics and Nature Inspired Computing (META 2012)*, Sousse, Tunisia, Oct. 27–31 2012.

[164] S. Varrette, J. Muszyński, and P. Bouvry. Cheating impact on distributed Evolutionary Algorithms over BOINC computations. In *Proc. of the 19th Int. Conf. on Security and Intelligent Information Systems (SIIS 2011)*, Warsaw, Poland, June 13–14 2011. Extended Abstract.

1.4 Dissertation Outline

The manuscript is organized as follows:

- The first part (Part I) presents the essential information required for reading the dissertation. In particular, the background together with some basic notions (linked to graph theory, distributed computing or Evolutionary Algorithms) are proposed.
- The second part of the thesis (Part II) covers the Cheating-Tolerance of Parallel and Distributed EAs in Desktop Grids and Volunteer Computing Systems (DGVCS's).
- The third part (Part III) concludes the manuscript and offers perspectives for future works.

We now briefly review the outline of the successive chapters of this manuscript.

Part I – Background and Basic Notions

Chapter 2 is mainly an extract of the most essential information from [162] regarding graph theory, and more precisely random graphs. It starts with an introduction of basic definitions for graphs and their metrics. After that, common models of random graphs, their properties and algorithms for their creation are presented.

Chapter 3 reviews distributed computing systems, focusing on their main properties, features and applications. Also, overlay networks are presented, as they determine communication and interaction patterns within distributed applications. Finally, an overview of failures, robustness and fault-tolerance techniques is proposed.

Chapter 4 is dedicated to the general characteristics and models of Evolutionary Algorithms (EAs). The generic scheme of an EA is presented, followed by an overview of different components building the algorithm. Finally, the underlying tools and approaches used to theoretically analyse EAs are reviewed.

Part II – Cheating-Tolerance of Parallel and Distributed EAs in DGVCS's

Chapter 5 starts with a discussion about sources of the most common faults present in DGVCS's, namely crash and cheating faults. Terms associated with volatility of such platforms are introduced, which are the main cause of crash failures. After that, cheating faults in DGVCS's are formally defined. This is followed by the state-of-the-art literature review about concrete EAs models for real-world, distributed executions with a discussion about their fault-tolerance and applicability in DGVCS's.

Chapter 6 presents one of the main contributions of the thesis. The theoretical model introduced in Chapter 4 is applied to the EA analysis in a hostile environment. Thanks to this, conditions for convergence (or non-convergence) toward valid solutions in the presence of cheaters are given. Finally, general guidelines for fault-tolerant centralised executions in DGVCS's are given.

Chapter 7 focuses on the second main contribution of this work: spatially-structured EAs and gossip protocols for Peer-to-Peer (P2P) executions. First, cheating-tolerance of distributed EAs is analysed from two perspectives: the evolution and the communication layers. The study is focused in particular on Evolvable Agent Model (EvAg) and its reference implementation based on Newscast. The required conditions and execution schemes allowing the computation to converge toward valid solutions despite malicious acts are developed. After establishing that the global success of the

optimisation may be affected only by tampering with the communication layer, the study is directed towards it. Due to the lack of a suitable replacement of Newscast for EvAg execution, cheating-tolerance of the protocol is analysed in details. As a result, a connectivity-splitting attack on the network is designed. It is experimentally demonstrated that the connectivity can be broken in a relatively short time, using very few naïve cheaters. Finally, this chapter is closed by a discussion and evaluation of proposed countermeasures, leading to a more robust solution.

At the end of the manuscript, a final part (Part III) features the chapters concluding the dissertation. At this occasion, some conclusions of the work performed are presented, together with outlines on perspectives and future work.
Figure 1.2 presents dependencies between the different chapters and therefore proposes a reading order for the thesis.

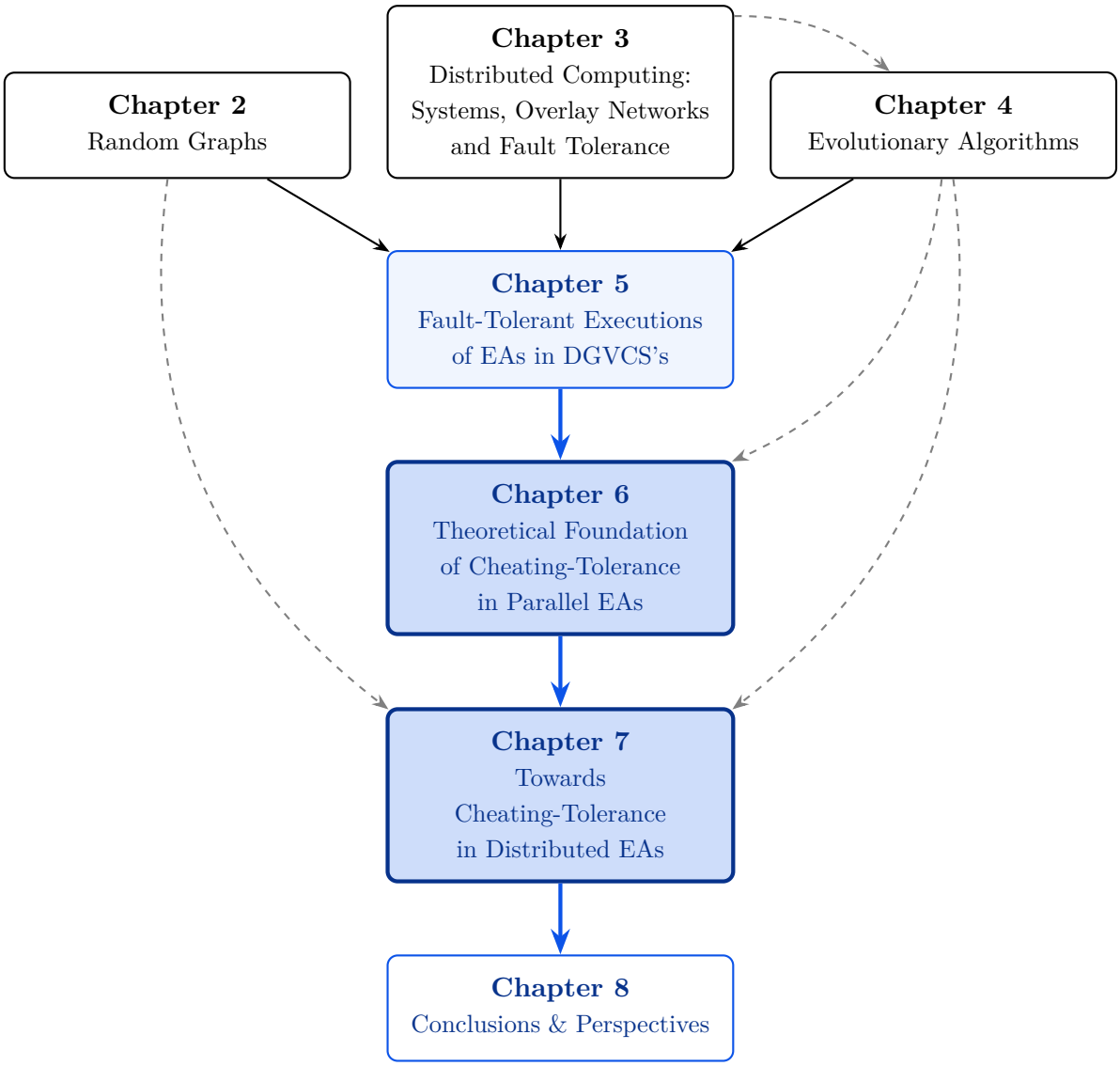


Figure 1.2: Graph of dependencies between the chapters of the thesis.

Part I

Background and Basic Notions

Chapter 2

Random Graphs

Contents

This chapter is mainly an extract of the most essential information from [162]. It begins with an introduction of basic definitions for graphs and their metrics. After that, common models of random graphs, their properties and algorithms for their creation are presented.

Essentially, a random graph of this type is created by a random addition of edges (hence the name). In details, the construction starts with a collection of n vertices. For each of the $\binom{n}{2}$ possible edges, an edge (u, v) is added with a probability p_{uv}. In a simplified case, for every pair of distinct vertices u and v, the probability p_{uv} is the same.

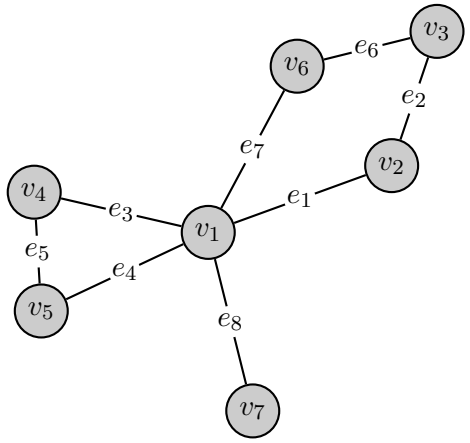
In the late 1950s, the theoretical foundations of random graphs were laid by Paul Erdös and Alfréd Rényi. After that, it was discovered that this construct could be used to describe many naturally occurring phenomena [162]. Since that, research in this area was expanded and applied to many fields — from neurology, through traffic management, to communication networks [162].

In the sequel, terms graphs and networks will be used interchangeably, the same applies to vertices and nodes.

2.1 Formalities

Intuitively a graph consists of a collection of vertices connected by edges, where each edge joins exactly two vertices (Definition 2.1.1, example — Figure 2.1).

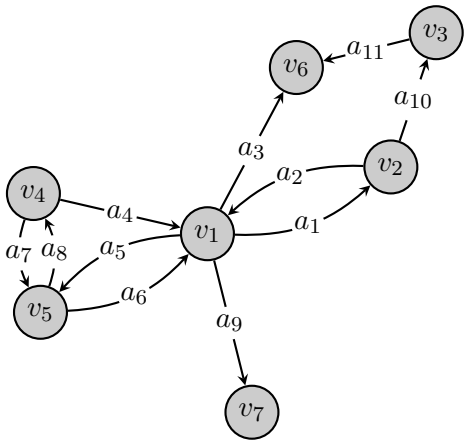
Definition 2.1.1 (Graph)**.** *A graph $G = (V, E)$ consists of a collection of vertices V and a collection of edges E. Each edge $e \in E$ joins two vertices $u, v \in V$, which is denoted by*



Graph:
$G = (V, E)$
Vertices:
$V = \{v_1, v_2, \ldots, v_7\}$
Edges:
$E = \{e_1, e_2, \ldots, e_8\}$

$e_1 = \langle v_1, v_2 \rangle$	$e_5 = \langle v_4, v_5 \rangle$
$e_2 = \langle v_2, v_3 \rangle$	$e_6 = \langle v_3, v_6 \rangle$
$e_3 = \langle v_1, v_4 \rangle$	$e_7 = \langle v_1, v_6 \rangle$
$e_4 = \langle v_1, v_5 \rangle$	$e_8 = \langle v_1, v_7 \rangle$

Figure 2.1: An example of an undirected graph.



Digraph:
$D = (V, A)$
Vertices:
$V = \{v_1, v_2, \ldots, v_7\}$
Arcs:
$A = \{a_1, a_2, \ldots, a_{10}\}$

$a_1 = \langle \overrightarrow{v_1, v_2} \rangle$	$a_6 = \langle \overrightarrow{v_5, v_1} \rangle$
$a_2 = \langle \overrightarrow{v_2, v_1} \rangle$	$a_7 = \langle \overrightarrow{v_4, v_5} \rangle$
$a_3 = \langle \overrightarrow{v_1, v_6} \rangle$	$a_8 = \langle \overrightarrow{v_5, v_4} \rangle$
$a_4 = \langle \overrightarrow{v_4, v_1} \rangle$	$a_9 = \langle \overrightarrow{v_1, v_7} \rangle$
$a_5 = \langle \overrightarrow{v_1, v_5} \rangle$	$a_{10} = \langle \overrightarrow{v_2, v_3} \rangle$

Figure 2.2: An example of a digraph.

*$e = \langle u, v \rangle$ and u, v are called its **end points**. If $e = \langle u, v \rangle$ then u, v are called **adjacent**, and edge e is said **incident** to vertices u and v.*

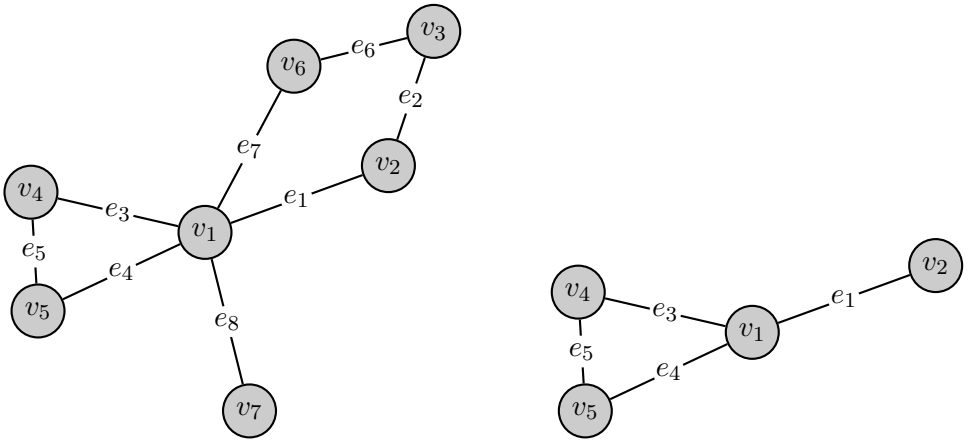
Sometimes for simplicity $V(G)$, respectively $E(G)$, will be used to describe the set of vertices, respectively edges, associated with a graph G.

If an edge e joins same vertices (i.e. $e = \langle v, v \rangle$), then e is called a **loop**. If two vertices may be connected by more than one distinct edge, then such graph is called **multigraph**. A graph without loops or multiple edges is called **simple**.

If the edges have associated direction, graph is described as directed or simply called — a digraph (Definition 2.1.2, example — Figure 2.2).

Definition 2.1.2 (Directed graph or digraph)**.** *A **directed graph** or **digraph** $D = (V, A)$ consists of a collection of vertices V and a collection of **arcs** A. Each arc $a = \langle \overrightarrow{u, v} \rangle$ joins vertex $u \in V$ to another (not necessarily distinct) vertex v. Vertex v is called the **head** of a, whereas u is its **tail**.*

At times it is beneficial to analyse undirected graph created from directed one (an underlying graph). Such construct is described in Definition 2.1.3. The graph G from Figure 2.1 is the underlying graph of the digraph D from Figure 2.2, end conversely — D is an orientation



(a) (b)

Figure 2.3: An example of a subgraph. In (a), a graph $G = (V, E)$ with $V = \{v_1, v_2, \ldots, v_7\}$ and $E = \{e_1, e_2, \ldots, e_8\}$. In (b) a subgraph $H = (V', E')$ of G induced by a vertex set $V' = \{v_1, v_2, v_4, v_5\}$ or by an edge set $E' = \{e_1, e_3, e_4, e_5\}$.

of G. This proved to be very often useful in directed networks which do not allow the plain evaluation of some metrics. For example, in case of distance statistics (see Section 2.2.3), if there is no directed path between some nodes, metrics become undefined or reach infinite values. In these cases, the notion of the **underlying graph** becomes handy. Information obtained from such analysis is inaccurate, but in practice serves well as an estimation to describe original graph's properties.

Definition 2.1.3 (Underlying graph and orientation)**.** *A graph $G(D)$ obtained from a digraph D by replacing each arc with its undirected counterpart is called an **underlying graph**. An **orientation** is a digraph $D(G)$ created by associating a direction with each edge of the graph G.*

Also, there is often an interest in the analysis of the surrounding of a vertex. In such cases, definitions of **neighbour**, **in-neighbour** and **out-neighbour** sets are required (see Definitions 2.1.4 and 2.1.5).

Definition 2.1.4 (Neighbour set)**.** *For any graph $G = (V, E)$ and a vertex $v \in V$, the set of vertices adjacent to v (other than v) is called **neighbour set** and is denoted by $N(v)$. Formally*

$$N(v) \stackrel{\text{def}}{=} \{u \in V \mid u \neq v, \exists_{e \in E}\, e = \langle u, v \rangle\}$$

Definition 2.1.5 (In- and out-neighbour sets)**.** *For a digraph $D = (V, A)$ and a vertex $v \in V$, the **in-neighbour set** $N^+(v)$ of v is composed of the adjacent vertices having an arc with v as its head. Likewise, the **out-neighbour set** $N^-(v)$ of v consists of the adjacent vertices having an arc with v as its tail. Formally*

$$N^+(v) \stackrel{\text{def}}{=} \left\{u \in V \mid v \neq u, \exists_{a = \langle \overrightarrow{u,v} \rangle}\, a \in A\right\}$$

$$N^-(v) \stackrel{\text{def}}{=} \left\{u \in V \mid v \neq u, \exists_{a = \langle \overrightarrow{v,u} \rangle}\, a \in A\right\}$$

The set of neighbours $N(v)$ of a vertex v is defined as $N(v) \stackrel{\text{def}}{=} N^+(v) \cup N^-(v)$.

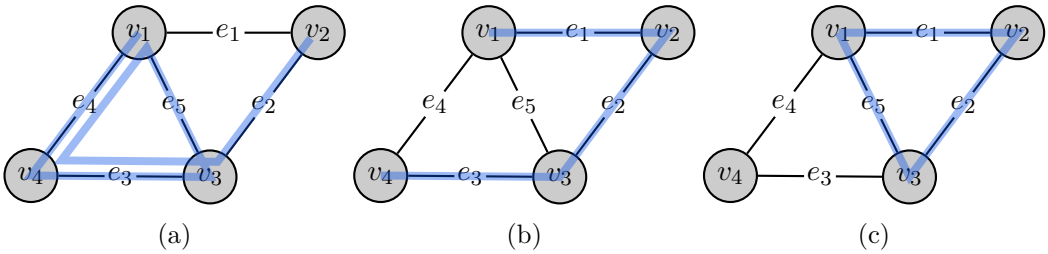


Figure 2.4: An illustration of a walk, a path and a cycle in a graph. In (a), a (v_1, v_2)-walk: $[v_1, e_4, v_4, e_3, v_3, e_5, v_1, e_4, v_4, e_3, v_3, e_2, v_2]$, (b) a path between v_1 and v_4: $[v_1, e_1, v_2, e_2, v_3, e_3, v_4]$ and (c) a cycle: $[v_1, e_1, v_2, e_2, v_3, e_5, v_1]$.

When only part of the graph is of interest and has to be analysed (beyond neighbour sets), notion of **subgraph** is required. Definition 2.1.6 introduces a way to determine, whether a given graph is a subgraph of the other. Definition 2.1.7 presents a method to create a subgraph given a subset of the vertex or edge set. An example of the introduced notions is depicted on Figure 2.3.

Definition 2.1.6 (Subgraph)**.** *A graph H is a **subgraph** of G, denoted by $H \subseteq G$, if $V(H) \subseteq V(G)$, $E(H) \subseteq E(G)$ and for all $e = \langle u, v\rangle \in E(H)$, we have that $u, v \in V(H)$.*

Definition 2.1.7 (Subgraphs induced by a vertex or an edge set)**.** *For a graph $G = (V, E)$ and a subset $V' \subseteq V$, the **subgraph induced by** V' has the vertex set V' and an edge set E' defined by*

$$E' \stackrel{\text{def}}{=} \{e \in E \mid e = \langle u, v\rangle \text{ with } u, v \in V'\}$$

*Similarly, if $E' \subseteq E$, the **subgraph induced by** E' has the edge set E' and a vertex set V' defined by*

$$V' \stackrel{\text{def}}{=} \{u, v \in V \mid \exists_{e \in E'}\, e = \langle u, v\rangle\}$$

The subgraph induced by V' or E' is denoted by $G[V']$ or $G[E']$, respectively.

Edges or arcs in a graph define immediate relation between vertices. If transitional linkage is required, definitions of walks, trials, paths and cycles in the network are used (Definitions 2.1.8 and 2.1.9, with a simple illustration on Figure 2.4).

Definition 2.1.8 (Walk, trial, path, cycle)**.** *Given a graph $G = (V, E)$, a (v_0, v_k)-**walk** in G is a sequence $[v_0, e_1, v_1, e_2, \ldots v_{k-1}, e_k, v_k]$, where $v_i \in V$ and $e_i \in E$. If $v_0 = v_k$, then such walk is called **closed**. A **trial** is a walk with all edges distinct, i.e. $\forall_{i \neq j}\, e_i \neq e_j$. A **path** is a trial with all vertices distinct, i.e. $\forall_{i \neq j}\, v_i \neq v_j$. A **cycle** is a closed trial in which all vertices except v_0 and v_k are distinct.*

Definition 2.1.9 (Directed walk, trial, path, cycle)**.** *For a digraph $D = (V, A)$, a **directed** (v_0, v_k)-**walk** in D is a sequence $[v_0, a_0, v_1, a_1, \ldots v_{k-1}, a_{k-1}, v_k]$, where $v_i \in V$, $a_i \in A$ and $a_i = \langle \overrightarrow{v_i, v_{i+1}} \rangle$. If $v_0 = v_k$, then such walk is called **closed**. A **directed trial** is a directed walk with all arcs distinct, i.e. $\forall_{i \neq j}\, a_i \neq a_j$. A **directed path** is a directed trial with all vertices distinct, i.e. $\forall_{i \neq j}\, v_i \neq v_j$. A **directed cycle** is a directed trial in which all vertices are distinct except for v_0 and v_k.*

2.2 Metrics

To compare graphs between each other, metrics have to be defined. Along with the definitions, the importance of presented measures in the analysis of real-world networks is discussed.

2.2.1 Vertex degree

Definition 2.2.1 (Vertex degree in undirected graphs)**.** *For a given graph $G = (V, E)$, $\delta(v)$ is a **degree** of a vertex $v \in V$, equal to the number of edges incident with v, where loops are counted twice. Formally*

$$\delta(v) \stackrel{\text{def}}{=} |\{e \in E \mid \exists_{u \in V}\, e = \langle u, v \rangle\}|$$

Definition 2.2.2 (Vertex in- and out-degrees in directed graphs)**.** *For a given digraph $D = (V, A)$, $\delta^+(v)$ is an **in-degree** of a vertex $v \in V$, equal to the number of incident arcs having v as their head. Likewise, $\delta^-(v)$ is the **out-degree** of a vertex v, equal to the number of incident arcs having v as their tail. Formally*

$$\delta^+(v) \stackrel{\text{def}}{=} |\{a \in A \mid \exists_{u \in V}\, a = \langle \overrightarrow{u, v} \rangle\}|$$

$$\delta^-(v) \stackrel{\text{def}}{=} |\{a \in A \mid \exists_{u \in V}\, a = \langle \overrightarrow{v, u} \rangle\}|$$

Despite its simplicity, vertex degree (Definitions 2.2.1 and 2.2.2) can play a surprisingly important role in the devising or analysing real-world networks [162]. The main function of this metric is the identification of key vertices in the graph — those nodes with a high value of the discussed measure. For example, the degree of a given node can help to estimate the amount of messages expected per time unit, i.e. the rate of incoming messages. This can help to avoid possible overloads in the network.

The vertex degree sequence can be used to obtain information about the structure of a graph. If most of the measurements are the same, the graph is more or less regular — all vertices have equal roles. If the sequence is very skewed — very few nodes have relatively high degree, then those vertices play the role of **hubs** in the network. Their removal could cause the graph to split [162].

Additionally, an important technique for analysing networks is computation of a **vertex degree distribution**, which shows how many vertices have certain degree. In practice, in-degrees are only taken into account [162]. For example, in case of social networks, the discussed distribution can show how important are different nodes. This also helps to get an impression of exactly how more important certain nodes are, by computing the ratio of in-degrees between different nodes.

Finally, **degree correlations** can be computed to determine which nodes are typically connected with each other. For example, in case of social networks, high-degree vertices are generally connected to each other, whereas in the technological networks, high-degree vertices are joined with low-degree ones [120, 162].

2.2.2 Connectivity

Analysing connectivity allows to asses if delivery of a message between two different nodes inside the network is possible. It also helps to identify critical nodes, which removal would

cause the graph to split into components. In case of communication protocols, their robustness can be evaluated by estimating the probability of the graph to split. Connectivity of the graph can be verified using graph traversal algorithms such as Breath-First Search (BFS) or Depth-First Search (DFS) [33, 162].

Notion of vertex and graph connectivity is introduced in the Definition 2.2.3. In case of digraphs, where direction of the edge matters, the approach for connectivity analysis differs. There, definitions of **strong and weak connectivity** are of interest (Definition 2.2.4).

Definition 2.2.3 (Connected vertices and graph)**.** *In a graph $G = (V, E)$, two vertices $u, v \in V : u \neq v$ are **connected** if there exists a (u, v)-path in G. G is **connected** if all pairs of distinct vertices are connected.*

Definition 2.2.4 (Strong and weak connectivity of a digraph)**.** *If there exists a directed path between every pair of distinct vertices from digraph D, then D is **strongly connected**. A digraph is **weakly connected** if its underlying graph is connected (see Definition 2.1.3).*

The whole graph does not have to be connected. It can consist of a collection of connected subgraphs, also called components – see Definition 2.2.5. In practice, if the graph is not connected, the notion of **giant component** is of interest – see Definition 2.2.6.

Definition 2.2.5 (Component)**.** *A **component** is a connected subgraph of G not contained in a different connected subgraph of G with more vertices or edges. The number of components of G is denoted as $\omega(G)$.*

Definition 2.2.6 (Giant component)**.** *A **giant component** is a component of G of the maximum size in terms of the number of vertices.*

2.2.3 Distance statistics

Another important class of graph metrics are distance statistics. The distance between two vertices in a graph is expressed as the length of the shortest path between them (Definition 2.2.7). If this information is computed for the whole graph, the importance of each of the vertices can be determined along with the partial information about the graph structure (Definition 2.2.8).

Definition 2.2.7 (Distance)**.** *For a graph $G = (V, E)$ (directed or not) and vertices $u, v \in V$, the **distance** between u and v is the length of the shortest (u, v)-path, denoted as $\mathrm{d}(u, v)$.*

Definition 2.2.8 (Eccentricity, radius, diameter)**.** *Given a connected graph $G = (V, E)$, let $\mathrm{d}(u, v)$ denote the distance between vertices u and v ($u, v \in V$). The **eccentricity** $\epsilon(v)$ of a vertex v in G tells how far the farthest vertex from v is positioned, formally:*

$$\epsilon(v) \stackrel{\text{def}}{=} \max \{\mathrm{d}(v, u) \mid u \in V\}$$

*The **radius** $\mathrm{rad}(G)$ indicates how dispersed the vertices are in the graph G and is defined as the minimum over all eccentricity values, formally:*

$$\mathrm{rad}(G) \stackrel{\text{def}}{=} \min \{\epsilon(v) \mid v \in V\}$$

*Finally, the **diameter** $\mathrm{diam}(G)$ describes the maximal distance in the graph, formally:*

$$\mathrm{diam}(G) \stackrel{\text{def}}{=} \max \{\mathrm{d}(u, v) \mid u, v \in V\}$$

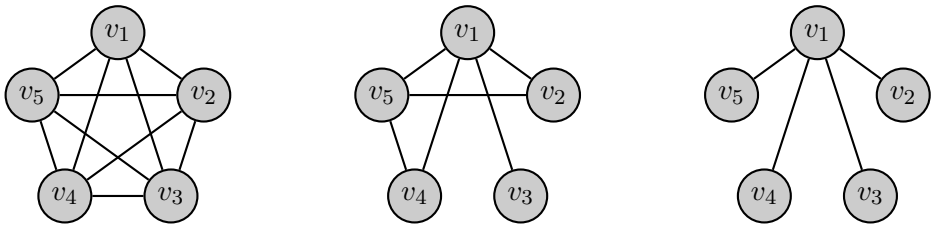


Figure 2.5: Example of clustering coefficients for nodes in different networks. In (a) $\mathrm{cc}(v_1) = \frac{2 \cdot 6}{4 \cdot 3} = 1$ (vertices from v_2 to v_5 — neighbours of v_1 — are fully connected), (b) $\mathrm{cc}(v_1) = \frac{2 \cdot 2}{4 \cdot 3} = 0.3$ and (c) $\mathrm{cc}(v_1) = 0$ (no edges between neighbours of v_1).

In practice, global statistics from Definition 2.2.8 are not powerful enough to discriminate among graphs [162]. Therefore, distribution of path lengths (in particular the **average distance between vertices**) is being used (Definition 2.2.9). For very large graphs (thousands of nodes), computation of the average path length can be costly and time-consuming [162]. In such cases, the **characteristic path length** is preferred, as there exist efficient techniques for its approximation [162] (in contrast to the previous metric).

Definition 2.2.9 (Average and characteristic path length)**.** *Let $G = (V, E)$ be a connected graph and $\overline{\mathrm{d}}(v)$ denote the average distance from a vertex v to any other vertex u in G:*

$$\overline{\mathrm{d}}(v) \stackrel{\mathrm{def}}{=} \frac{1}{|V| - 1} \sum_{u \in V, u \neq v} \mathrm{d}(v, u)$$

*The **average path length** $\overline{\mathrm{d}}(G)$ of a graph G is defined as*

$$\overline{\mathrm{d}}(G) \stackrel{\mathrm{def}}{=} \frac{1}{|V|} \sum_{v \in V} \overline{\mathrm{d}}(v)$$

*The **characteristic path length** of G is defined as the median over all $\overline{\mathrm{d}}(u)$.*

2.2.4 Clustering Coefficient

Clustering coefficient (introduced in [176]) measures for a given vertex v, to what extent its neighbours are neighbours of each other, i.e. to what extent vertices adjacent to v are also adjacent to each other. Therefore, this statistic gives some information about the graph structure from the local point of view. For example, high value of the clustering coefficient indicates existence of **communities** in the graph — interconnected groups of nodes with dense links within them and sparse in between [162]. Definition 2.2.10 formalises this metric for a vertex from a simple connected, undirected graph (examples illustrated on Figure 2.5); Definition 2.2.11 — for a vertex from a simple connected digraph and Definition 2.2.12 for both types of graphs (as a global statistic).

Definition 2.2.10 (Clustering coefficient for a vertex from an undirected graph)**.** *Given a simple connected, undirected graph $G = (V, E)$ and a vertex $v \in V$ with a neighbour set $N(v)$, let $n_v = |N(v)|$ and $m_v = |E(G[N(v)])|$ (the number of edges in the subgraph induced by neighbour set of v). The **clustering coefficient** $\mathrm{cc}(v)$ for the vertex v with degree $\delta(v)$*

is defined as

$$\mathrm{cc}(v) \stackrel{\mathrm{def}}{=} \begin{cases} m_v / \binom{n_v}{2} = \frac{2 \cdot m_v}{n_v (n_v - 1)} & \text{if } \delta(v) > 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 2.2.11 (Clustering coefficient for a vertex from a digraph)**.** *Given a simple connected digraph $D = (V, A)$ and a vertex $v \in V$ with a neighbour set $N(v)$, let $n_v = |N(v)|$ and $m_v = |A(G[N(v)])|$ (the number of arcs in the subgraph induced by neighbour set of v). The **clustering coefficient** $\mathrm{cc}(v)$ for the vertex v with the degree $\delta(v) = \delta^+(v) + \delta^-(v)$ is defined as*

$$\mathrm{cc}(v) \stackrel{\mathrm{def}}{=} \begin{cases} m_v / \left(2 \cdot \binom{n_v}{2}\right) = \frac{m_v}{n_v (n_v - 1)} & \text{if } \delta(v) > 1 \\ \text{undefined} & \text{otherwise} \end{cases}$$

Definition 2.2.12 (Average clustering coefficient)**.** *For a simple connected graph $G = (V, E)$, the **average clustering coefficient** $\mathrm{CC}(G)$ is defined as*

$$\mathrm{CC}(G) \stackrel{\mathrm{def}}{=} \frac{1}{|V'|} \sum_{v \in V'} \mathrm{cc}(v)$$

where $V' = \{v \in V \mid \delta(v) > 1\}$.

From the global point of view, there also exist an alternative definition for the clustering coefficient [121]. It is based on the number of **triples** and **triangles** in a graph. For a given vertex $v \in V$ from a simple, connected graph $G = (V, E)$:

- **a triangle at** v is a complete subgraph of G with exactly three vertices (including v),
- **a triple at** v is a subgraph of exactly three vertices and two edges incident to v.

The number of triangles and triples at v are denoted as $n_\Delta(v)$ and $n_\Lambda(v)$ respectively. Globally for the graph G, $n_\Delta(G)$ denotes a number of distinct triangles and $n_\Lambda(G)$ — triples. To avoid confusion (as emphasised in [121]), this statistic will be called a **graph transitivity** and is formalised in Definition 2.2.13.

Definition 2.2.13 (Graph transitivity)**.** *For a simple, connected graph G, the **graph transitivity** $\tau(G)$ is the ratio $n_\Delta(G) / n_\Lambda(G)$.*

Sometimes (for instance in the case of a social network analysis) the **network density** is used as the clustering coefficient [71, 162, 175]. This measure indicates to what extent a graph is complete — see Definition 2.2.14.

Definition 2.2.14 (Graph density)**.** *The **graph density** $\rho(G)$ for a simple, undirected graph $G = (V, E)$ is defined as $m / \binom{n}{2}$, where $m = |E|$ and $n = |V|$.*

As visible, there are different approaches used in the literature to measure the clustering of a given graph. The usage of the graph density (Definition 2.2.14) is mainly limited to the social network analysis. On the other hand, the average clustering coefficient and the graph transitivity are very often simply referred to as the **clustering coefficient of a graph**, without specifying which definition is in use, even though they are not equal [121, 162]. The main difference is that the first one "*tends to weight the contributions of low-degree vertices more heavily*" [121]. From the analytical point of view, it is easier to compute the graph transitivity, whereas the average clustering coefficient is easily calculated on a computer. Therefore, the second one is widely used in numerical studies and data analysis [121]. In this work, Definition 2.2.12 will be used (the average clustering coefficient).

2.2.5 Centrality

The last discussed here, but not the least significant metric is **centrality** [119]. It allows identifying "*important*" vertices in the graph. Of course, importance of a node depends on what is actually modelled. For example, if the graph is used to analyse a communication network, importance of a vertex may be defined as a number of the shortest paths in which it belongs [162]. This might serve as an indication of the expected workload regarding processing and forwarding messages [162].

The basic notion of the centrality is a centre of a graph — vertices whose eccentricity is equal to the radius of the graph (Definition 2.2.15). Following this reasoning, one can say that a vertex is at the centre of a graph when its distance from all other vertices is minimal. Therefore, centrality of a vertex could be defined using its eccentricity — Definition 2.2.16.

Definition 2.2.15 (Centre of a graph)**.** *Let $G = (V, E)$ be a (strongly) connected graph. The* ***centre*** $\mathrm{C}(G)$ *of a graph G is the set of vertices with minimal eccentricity.*

$$\mathrm{C}(G) \stackrel{\text{def}}{=} \{v \in V \mid \epsilon(v) = \mathrm{rad}(G)\}$$

Definition 2.2.16 ((Eccentricity based) vertex centrality)**.** *For a (strongly) connected graph $G = (V, E)$, the* ***(eccentricity based) vertex centrality*** $c_E(v)$ *of a vertex $v \in V$ is defined as $c_E(v) \stackrel{\text{def}}{=} 1/\epsilon(v)$.*

It is visible from these formalisations that the farther a vertex is from the centre, the lower the value of the mentioned metric. Where centrality reaches its maximum for all the nodes located at the centre of the graph.

Sometimes, it is more useful to determine how close a vertex is to all other vertices. This requires taking into account distances to all the nodes from a given one. Such metric is called the **closeness** and is introduced in Definition 2.2.17.

Definition 2.2.17 (Closeness)**.** *Let $G = (V, E)$ be a (strongly) connected graph. The* ***closeness*** $c_C(v)$ *of a vertex $v \in V$ is defined as*

$$c_C(v) \stackrel{\text{def}}{=} \frac{1}{\sum_{u \in V} \mathrm{d}(v, u)}$$

Closeness comparison between vertices of different graphs may not be very useful [162]. This metric is biased by the graph size — the value for a fixed vertex decreases when more nodes are added.

There exists also another metric describing "*importance*" of a vertex — **betweenness centrality** (Definition 2.2.18). In this approach, a node is considered as "*important*" if it lies on many shortest paths connecting two other nodes. Removal of such vertex may increase the cost of the connectivity between other vertices — different (possibly longer) paths will have to be followed [162].

Definition 2.2.18 (Betweenness centrality [162])**.** *Consider a simple, (strongly) connected graph $G = (V, E)$. Let $S(x, y)$ be the set of the shortest paths between two vertices $x, y \in V$, and $S(x, v, y) \subseteq S(x, y)$ the ones that pass through a vertex $v \in V$. The* ***betweenness centrality*** $c_B(v)$ *of a vertex v is defined as*

$$c_B(v) \stackrel{\text{def}}{=} \sum_{x \neq y} \frac{|S(x, v, y)|}{|S(x, y)|}$$

2.3 Classical random graphs

In 1959, Paul Erdös and Alfréd Rényi introduced two "*classical*" models for random networks [50] (often called **Erdös-Rényi networks** or simply **ER random graphs**). In both of them, only undirected edges are taken into account. First approach is based on two parameters: the number of vertices — n and the probability — p. Using those variables, an undirected graph $G_{n,p}$ with n nodes is created, where two (distinct) vertices are connected by an edge with a probability p (independently). In the second approach, instead of the parameter p, number of edges — M is considered. This yields an undirected graph $G_{n,M}$ in which M edges are incident to randomly chosen pairs of vertices. The main difference between both models is that the first one ($G_{n,p}$) have no loops or multiple edges between two distinct vertices, where in the second ($G_{n,M}$) — there is no such restrictions. Even though, such constraints are often introduced in practice, by limiting the choice to M distinct edges from the $n(n-1)/2$ possibilities [45].

In this work, only simple graphs are of interest. Additionally, the more common approach is to study the first version of the model, as the second one introduces small dependencies caused by picking a fixed number of edges [45]. Therefore, the approach based on the probability p will be considered from now on. In this context, $\mathrm{ER}(n,p)$ will be used to denote the set of all $G_{n,p}$ graphs.

There are four main metrics used to describe ER random graphs: the degree distribution, the average path length, the average clustering coefficient and the connectivity.

The vertex degree follows a binomial distribution. For a given node v from an $\mathrm{ER}(n,p)$ graph, let $\mathrm{P}[\delta(v)=k]$ denote the probability that the degree of v is k. There are $\binom{n-1}{k}$ possibilities for choosing k different vertices to be adjacent to v. A vertex v has exactly k neighbours with the probability equal to $p^k \cdot (1-p)^{n-1-k}$, therefore

$$\mathrm{P}[\delta(v)=k] = \binom{n-1}{k} p^k (1-p)^{n-1-k}$$

As this probability applies to all vertices of an $\mathrm{ER}(n,p)$ graph, it is clear that the vertex degree can be treated as a random variable δ. From this, the mean vertex degree is thus computed as

$$\overline{\delta} \stackrel{\mathrm{def}}{=} \mathrm{E}[\delta] \stackrel{\mathrm{def}}{=} \sum_{k=1}^{n-1} k \cdot \mathrm{P}[\delta = k]$$

Therefore, the vertex degree of an arbitrarily chosen vertex from and $\mathrm{ER}(n,p)$ is $p(n-1)$ (for the proof see [162]).

For large, connected random graphs $G \in \mathrm{ER}(n,p)$, the average path length can be estimated [57] as

$$\overline{\mathrm{d}}(G) = \frac{\ln(n) - \gamma}{\ln(pn)} + 0.5$$

where γ is the Euler constant (approximately equal to 0.5772).

The clustering coefficient of a vertex v is computed as the fraction of edges found between its neighbours and the maximum number of possible edges between them. From that, it is clear that the expected value of the clustering coefficient is equal to p (for the details see [162]). Consequently, for any graph $G \in \mathrm{ER}(n,p)$, we have $\mathrm{CC}(G) = p$.

In their original work, Erdös and Rényi, discovered that the appearance of many properties depends on a sharp threshold [45, 50]. These transitions are conditioned by the value of

$p = 1/n$. $\mathrm{ER}(n,p)$ graphs have different properties below, above and near this critical value [45].

Probability p influences the most connectivity and as a consequence — an emergence of a giant component. For large graphs ($n \to \infty$), if $p = c/n$ and $c < 1$, then most of the connected components of a graph are small, where the largest has only $\mathrm{O}(\log n)$ vertices. On the other hand, if $c > 1$ there is a constant $\Theta(c) > 0$ that the size of the largest component[1] is $\sim \Theta(c)\, n$ and the second one has $\mathrm{O}(\log n)$ vertices. The **threshold for connectivity** is $p = (\log n)/n + \mathrm{O}(1)$ — isolated vertices disappear and all the nodes belong to a single component. For details of all the above estimations see [45].

2.4 Small-world graphs

Erdös-Rényi graphs have small diameters, but have very few triangles [45]. Watts and Strogatz proposed a new model with a small diameter and a positive density of triangles (i.e. with a low average path length and a high clustering coefficient) — **small-world network**. It found its main application in the area of social networks, where the triangular relations are present very often. For instance, if A and B are friends and A and C are friends, then it is highly probable that B and C are also friends. Similarly, for the low average path length, the experiment of Stanley Milgram from 1967 (sending letters to a specific person, by passing it to people which the subjects know personally) revealed the resemblance of a social network to a small-world graph.

The construction of the original model proposed by Watts and Strogatz [176] starts from a regular lattice (e.g. ring lattice) with n vertices and k edges per vertex. After that, some edges are rewired at random with probability p. The mentioned rewiring operation is a simple replacement of each edge (u, v) with an edge (u, w), where w is a randomly chosen node from the set of vertices other than u, and such that (u, w) does not already belong to the edge set of the new (modified) graph. Described approach allows the "*tuning*" of the network between regularity ($p = 0$) and disorder ($p = 1$) [176] (see Figure 2.6). The set of all such graphs will be denoted as $\mathrm{WS}(n, k, p)$.

It is required that $n \gg k \gg \ln(n) \gg 1$ to guarantee connectivity between the nodes [176]. If this condition is met, then it is possible to estimate the asymptotic behaviour of the average path length and the average clustering coefficient with the probability p converging to its extremes. For two given graphs $S \in \mathrm{WS}(n, k, p)$ and $R \in \mathrm{ER}(n, p)$:

- if $p \to 0$, then $\overline{\mathrm{d}}(S) \sim n/2k \gg 1$ and $\mathrm{CC}(S) \sim 3/4$,
- if $p \to 1$, then $\overline{\mathrm{d}}(S) \approx \overline{\mathrm{d}}(R) \sim \ln(n)/\ln(k)$ and $\mathrm{CC}(S) \approx \mathrm{CC}(R) \sim k/n \ll 1$.

It is visible in Figure 2.6 that the regular lattice (a) is a highly clustered, "large world" — since the average path length grows linearly with the graph size n. Where the random network (c) is poorly clustered, the small world one (b) sees its average path length $\overline{\mathrm{d}}(G)$ growing logarithmically with n [176].

However, the transition between both described states is not linear. As has been experimentally tested for a family of $\mathrm{WS}(n, k, p)$, there is a broad interval of p over which the average path length is almost as small as for random graphs, yet the clustering coefficient is of greater order ($\mathrm{CC}(S) \gg \mathrm{CC}(R)$) [176].

Nevertheless, the Watts-Strogatz model raises issues with analytic treatment of general cases — as emphasized by Newman and Watts in [122]. First of all, distribution of the rewired

[1] $X_n \sim b_n$ (X_n on the order of b_n), means that X_n is equal to b_n asymptotically, i.e. $\frac{X_n}{b_n} \to 1$ when $n \to \infty$.

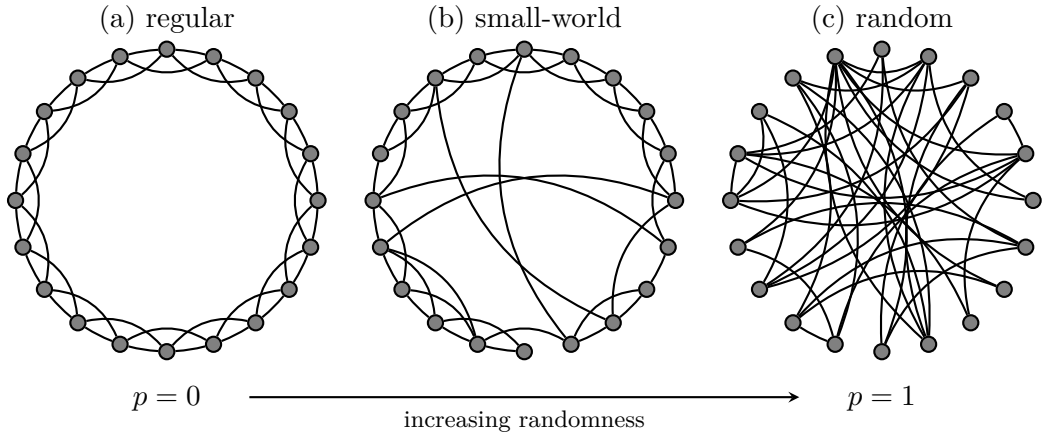


Figure 2.6: Parameter "tuning" in the Watts-Strogatz model. The initial network consists of $n = 20$ vertices, each connected to $k = 4$ nearest neighbours. With the increase of the parameter p (probability of rewiring the edge), increases the randomness — from a regular ring lattice (the left figure, $p = 0$) to a random graph (the right figure, $p = 1$).

edges is not completely uniform (e.g. multiple edges between two vertices are prohibited), therefore average over different realisations of the randomness is hard to perform. Secondly, there is a finite probability that the graph will be split during the process of rewiring. Therefore, average distance between pairs of vertices of the graph and number of quantities and expressions are poorly defined.

To circumvent these problems, an improved small-world model was proposed in [122]. Starting from a regular lattice, instead of rewiring each edge with a probability p, short-cuts are added between pairs of vertices chosen uniformly at random. There is also no prohibition on the existence of loops or multiple edges between two vertices. To preserve compatibility with the results of Watts and Strogatz, short-cuts are added with probability p for each existing edge in the original network. This model is equivalent to the Watts-Strogatz model for small p, whilst being better behaved when p becomes comparable to 1 (for details see [122]). As we will see later in the manuscript (in Chapter 4 and Appendix B), this construction was used in the expected runtime analysis of an elitist parallel Evolutionary Algorithm based on an island model executed over such a small-world network.

2.5 Scale-free graphs

The last type of random graphs described here are **scale-free networks**. This model was introduced by Albert-László Barabási and Réka Albert [18]. Scale-free graphs capture properties of many real-world networks such as World Wide Web links, biological networks or actor collaborations. The key property is the degree distribution of vertices following power laws[2]: $\mathrm{P}[\delta(v) = k] \sim k^{-\alpha}$ as $k \to \infty$ for $\alpha > 1$ called the **scaling exponent**. Which implies that there are a few high-degree nodes (hubs) and that the number of nodes with a high degree decreases exponentially.

Opposite to the previously described models, scale-free graphs can be constructed only through a **growth process** combined with a **preferential attachment** [162, 169]. The

[2]Sometimes simply written as $\mathrm{P}[k] \propto k^{-\alpha}$.

Algorithm 2.1: Construction of a Barabási-Albert random graph.

Data: $G_0 = (V_0, E_0) \in \mathrm{ER}(n, p)$
n — number of vertices to add
$v_1, v_2, \ldots, v_n$ — vertices to add
m — number of edges to add, $m \leq |V_0|$
Result: a Barabási-Albert random graph
for $i \leftarrow 1$ **to** n **do**
  $V_i \leftarrow V_{i-1} \cup \{v_i\}$;
  $V' \leftarrow V_{i-1}$;
  $E' \leftarrow \emptyset$;
  while $V' \neq \emptyset$ **and** $|E'| < m$ **do**
    choose a vertex u from V' with a probability $\mathrm{P}[\text{select } u] = \delta(u) / \sum_{w \in V'} \delta(w)$;
    $V' \leftarrow V' \setminus \{u\}$;
    $E' \leftarrow E' \cup (v_i, u)$;
  $E_i \leftarrow E_i \cup E'$;

method is detailed in Algorithm 2.1. The process starts with a (relatively small) ER random graph, which is one of the input parameters of the algorithm. At each step of the main **for** loop, new node (v_i) is added into the vertex set. After that, m edges are created between v_i and randomly chosen, unique vertices from the vertex set from previous step, where the probability of choosing a node is proportional to its degree. It is imperative that m is smaller or equal to the size of the original vertex set. The graph created this way is called a **Barabási-Albert random graph** or a **BA graph**. A family of such networks will be denoted as $\mathrm{BA}(n, n_0, m)$.

The connectivity of a BA random graph (being a result of Algorithm 2.1) depends on if the input network was connected (G_0). As visible from the algorithm, none of the original vertices or edges are changed. Additionally, the vertex being added in a current step of the **for** loop is connected to the graph from the previous step.

The vertex degree, as mentioned in the introduction of the model, follows the power law. For a given node v from an $\mathrm{BA}(n, n_0, m)$ graph, let $\mathrm{P}[\delta(v) = k]$ denote the probability that the degree of v is k, then

$$\mathrm{P}[\delta(v) = k] = \frac{2m(m+1)}{k(k+1)(k+2)} \propto \frac{1}{k^3}$$

as detailed in [162,169]. Taking a similar approach to the one used in Section 2.3 and treating the vertex degree as a random variable δ, it can be computed that the average vertex degree $\overline{\delta}$ is equal to $2m$.

The average path length for a graph $G \in \mathrm{BA}(n, n_0, m)$ can be estimated by [57]:

$$\overline{\mathrm{d}}(G) = \frac{\ln(n) - \ln(m/2) - 1 - \gamma}{\ln(\ln(n)) + \ln(m/2)} + 1.5$$

where γ is the Euler constant.

In case of scale-free networks, the analytical expression that estimates the average clustering coefficient has not yet been found [162]. For a graph $G \in \mathrm{BA}(n, n_0, m)$, Bállobas and

Riordan showed [17] that

$$\mathrm{CC}(G) \sim \frac{m-1}{8}\frac{(\log n)^2}{n}$$

2.6 Summary

This chapter introduced basic graph theory and metrics used to compare graphs between each other. Fundamental models of random graphs were presented, including: ER random graph, Watts-Strogatz small-world model, and scale-free Barábasi-Albert graph.

We use extensively all metrics defined here in the later chapters, especially in Chapter 5 and 7. As we mentioned earlier in the text, many natural phenomena and technological creations used in communication exhibit properties of random graphs. Sometimes, these characteristics are highly required. For instance, many Peer-to-Peer (P2P) networks are specifically designed to share properties with a random or a small-world graph. As a result, one can provide some assurances about the communication performance, resilience to network failures, etc.

Chapter 3

Distributed Computing: Systems, Overlay Networks and Fault Tolerance

Contents

Distributed computing refers to solving large-scale computational problems by their division into many tasks, each of which are executed by multiple networked and autonomous devices forming a **distributed system**.

This chapter starts with an overview of distributed systems, introducing their main properties, features and applications. After that, overlay networks are presented, as they determine communication and interaction patterns within distributed applications. Overview of failures, techniques to tolerate them and a definition of system robustness are closing the chapter.

3.1 Distributed systems

Distributed computing can also be defined as a field of computer science studying **distributed systems**. The main design goals of such systems can be grouped into two areas: HPC and High Throughput Computing (HTC). HPC refers to emphasis on the raw speed performance measured in FLOPS[1], associated with large scale scientific and engineering computations [77]. In case of HTC, the performance goal is shifted towards high throughput or the number of tasks completed per unit of time, which is characteristic for business and web services applications [77].

3.1.1 Clusters

Clusters (or **supercomputers**) were initially created to provide large collection of compute, storage and network resources for scientific and engineering applications. A typical build of such a system consists of homogeneous software and hardware [34,77]. Stand-alone, dedicated computers forming a cluster are located in proximity to each other [34]. The machines are interconnected in a low-latency, high-bandwidth network (such as Gigabit Ethernet, Myrinet, InfiniBand, etc.) [77]. This design is used to provide a single, integrated HPC environment with centralised management and control [34, 77].

3.1.2 (Cluster-based) Computing Grids

With increasing needs for compute and storage resources, an idea of a **grid** emerged. Such a system consists of loosely interconnected clusters from geographically dispersed sites [77]. Typically, it is a heterogeneous platform created to aggregate and share resources.

A grid is managed by a multi-institutional organisation [147]. Even so, each cluster owner retains control over owned resources, which is reflected in different local policies and software choices [147]. Because of this management model, a grid is sometimes called a **federation of clusters**.

Many national and international grids are built in various parts of the world with support and funding from governments (mainly) and industry. A local example is Grid'5000 [69] — initially a French platform spanning nine sites, established for research purposes. At the time of writing, the project is expanding (11 sites) in the international direction and the UL is one its active member. Similarly, the successful TeraGrid [34, 77] in the United States (11 sites, ended in 2011) has its continuation in a new venture supported by 17 institutions — Extreme Science and Engineering Discovery Environment (XSEDE) [180].

3.1.3 Desktop Grid and Volunteer Computing System (DGVCS)

The first distributed computing project aiming to utilise idle compute resources was created in 1978 [149]. The "worm" programs developed by Xerox Palo Alto Research Centre were able to span machine boundaries and replicate themselves in idle computers connected to a Local Area Network (LAN) [149]. Tests were conducted in a homogeneous environment consisting of 100 Xerox Alto Personal Computers (PCs). The success of this project laid the foundation for **opportunistic computer grids**, using non-dedicated resources.

[1] FLOPS — **FL**oating point **O**perations **P**er **S**econd

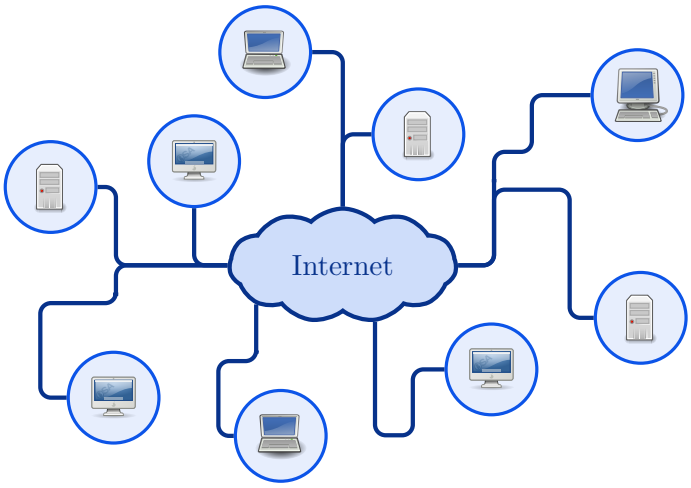


Figure 3.1: Example of a DGVCS implementation.

In the (current) literature on the distributed systems [34,77], **DGVCS's** are not presented in a separate category, but in combination with (cluster-based) grids. However, there are many differences between these schemes (like availability, level of security, hardware class, etc.), which aids the clear distinction.

Typically, a **desktop grid** is composed of heterogeneous, economical and partially available computers (PCs, workstations, etc.) [19]. An example is depicted in the Figure 3.1. Depending on the geographic dispersion of the machines, the system can be **local** or **global**. Usage of resources shared by a computer (connected to the desktop grid) may be conditioned on its state, e.g. a task execution is permitted only when the machine is **idle**[2] or utilisation of some resource is below a certain level. Finally, the system can be **private** or **public**. **Private desktop grids** are operated within an organisation (like a company, laboratory, university, etc.), whereas public ones rely mostly on **volunteer** (i.e. user-contributed) resources. In order to attract volunteers, system owners offer various incentives based on contributions, such as rewards, credit points, hall of fame, etc. In view of the above, a definition of **volunteer computing** can be given as a type of distributed computing with tasks executed in a **volunteer computing system** (global, public desktop grid).

DGVCS's are primarily used to support large-scale scientific projects. The biggest advantage of these systems is the reduction of costs associated with purchase and maintenance of dedicated hardware and physical space [19]. For instance, a private desktop grid can be built in a given organisation using owned PCs, laptops and workstations. Of course, a simple interconnection of these computers is not enough and additional means are needed. The most popular middleware software used to build such platforms is Condor [99] (renamed to HTCondor [76] in 2012), which is a specialised workload management system for batch processing.

In contrast to the cluster-based grids providing HPC environments, desktop-based solutions deliver HTC platforms required in applications focused on processing huge volumes of data. Volunteer computing projects SETI@home [9] and Folding@home [56] are well known, widely covered in media examples. First one is used to search for extra-terrestrial intelligence, the second — for disease research that simulates protein folding, computational drug

[2] A computer can be considered as idle when its CPU is in the idle state or when its user is inactive (the mouse and keyboard are idle).

design and other types of molecular dynamics. Thanks to the publicity, they have the largest number of volunteers. Nevertheless, data-intensive projects are not the only application area of desktop grids. GIMPS [60] (Great Internet Mersenne Prime Search) is one such instance, as it mainly requires high CPU power. All three examples (and many alike) have one thing in common: use of dedicated hardware causes a lack of their economic viability due to the scale and required resources [19]. This does not mean, that the cost disappears when DGVCS's is used, as it is simply passed on the users or donor organisations.

The aforementioned volunteer computing projects belong to the group of **single-purpose** DGVCS's. There are also general solutions (i.e. **general-purpose** DGVCS's) like HTCondor [76] (described earlier), Distributed.net [43] and BOINC [7]. Distributed.net is the creator of the Internet's first general-purpose distributed computing system, maintaining projects related to cryptography. BOINC was created in response to security issues associated with malicious users, identified in SETI@home. It is an open source project facilitating development of multi-purpose DGVCS's, providing a complete tool set for creation, operation and maintenance of public scientific projects.

Developers of DGVCS's have to cope with more challenges, than in the case of the previously described schemes. First, the **heterogeneity** of the machines composing the system is much more diverse and should be handled transparently [19]. Donated computers may have completely different hardware and software configurations (including various versions, vendors, tweaks, custom modifications etc.). This alone can lead to any unforeseen complications, conflicts and errors. On top of that, the network connection of each machine may be characterised by different bandwidth and latency, which has to be taken into account. Furthermore, volunteer resources are extremely **volatile**, so availability changes have to be gracefully handled. Ensuring security of the computers in a private desktop grid is much easier, than in the public systems. If the organisation is in control of the machines, proper usage policies may be introduced, which is very difficult in volunteer-based solutions. Finally, problems related to correctness have to be solved, which is particularly challenging in public systems. It is so because volunteers are essentially anonymous, which renders them unaccountable for their actions.

3.1.4 The Cloud Computing Paradigm

Cloud Computing (CC) systems emerged from commercial sector focused on enterprise applications, where consumers use and pay for what they need on the Internet [147]. Essentially, they implement the vision of computing as a utility through offering **"everything as a service"** [34]. Clouds are heavily dependant on dynamically scalable and often **virtualised resources**, which provide on-demand, instant and elastic environment at the scale and reliability of a data centre. From the hardware point of view, clouds are built on top of cluster-based grids. Therefore, these systems are often confused with grids themselves. However, there is a major difference between them: in clouds, a resource is leased to a consumer with the full control over it. This is supported by an underlying hypervisor isolating other resources securely. Whether clouds can provide enough performance and speed in computation, storage and networking for HPC applications, is a difficult question to answer and heavily depends on the concrete cloud realisation [167].

Cloud services come from main elements in Information Technology — software runs on a platform using an infrastructure. This is reflected in different pay-as-you-go, layered service models: **IaaS**, **PaaS** and **SaaS**. These layers are illustrated in the Figure 3.2 and are now detailed.

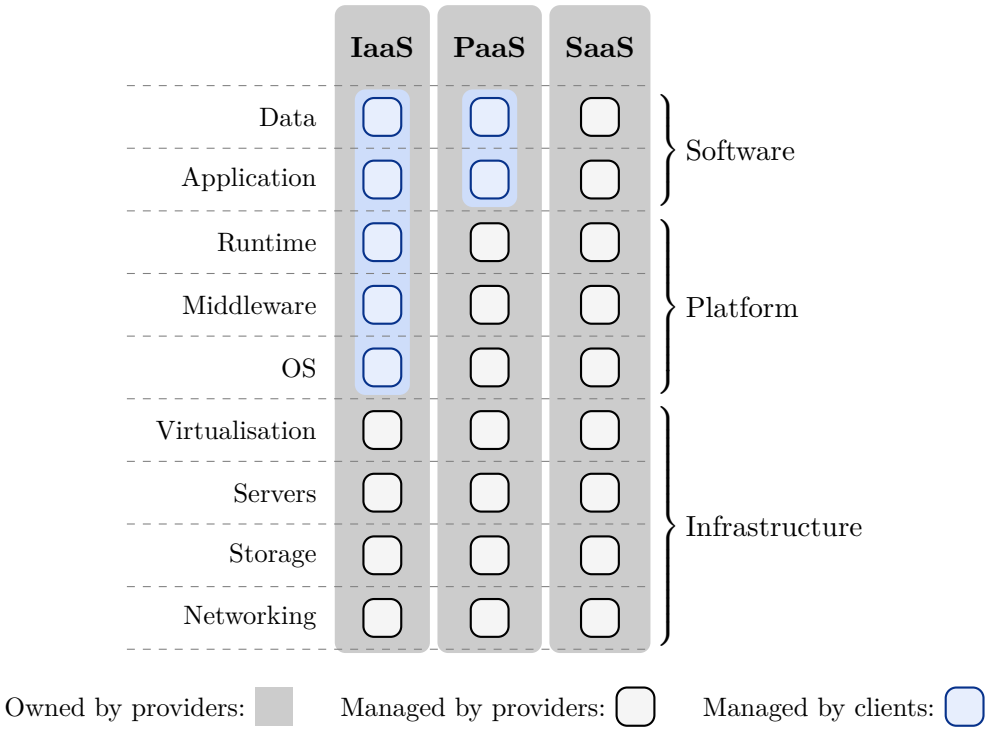


Figure 3.2: Cloud service models: IaaS, PaaS, and SaaS. On the left all elements building the system are enumerated, on the right — their membership in the classical application model is marked. Parts managed by the clients and providers are appropriately selected.

3.1.4.1 Infrastructure as a Service (IaaS)

The bottom (IaaS) layer is based on different Virtualisation Technologies (VTs) implemented on physical hardware. Its role is to put together infrastructures consisting of computing, storage, and networking resources demanded by users [77]. This is achieved by deployment and execution of multiple Virtual Machines (VMs), running guest Operating Systems (OS's) — both usually selected by the customers. The users do not control or manage the underlying cloud infrastructure [77]. Instead, they can supervise the OS's, storage and deployed applications [77]. Sometimes, customers receive control over selected networking components [77]. There are many commercial IaaS providers, for example: Amazon EC2 [6], GoGrid [61], Rackspace [132]. It is also possible to build private clouds using one of many available toolkits like Nimbus [123], OpenNebula [126], Cumulus [35], EUCALYPTUS [51], openQRM [127], OpenStack [128] etc.

3.1.4.2 Platform as a Service (PaaS)

PaaS is the middle layer of cloud services. It enables development, deployment, and management of user-build applications using provisioned, well-defined, virtualised platform [77]. PaaS is oriented on application lifecycle, hence customers use predefined infrastructures [77]. This includes also available software tools, programming languages and libraries. Such environments usually differ between providers. For instance, Google App Engine [67] let customers run programs in a limited version of Python or Java with access to Google's database;

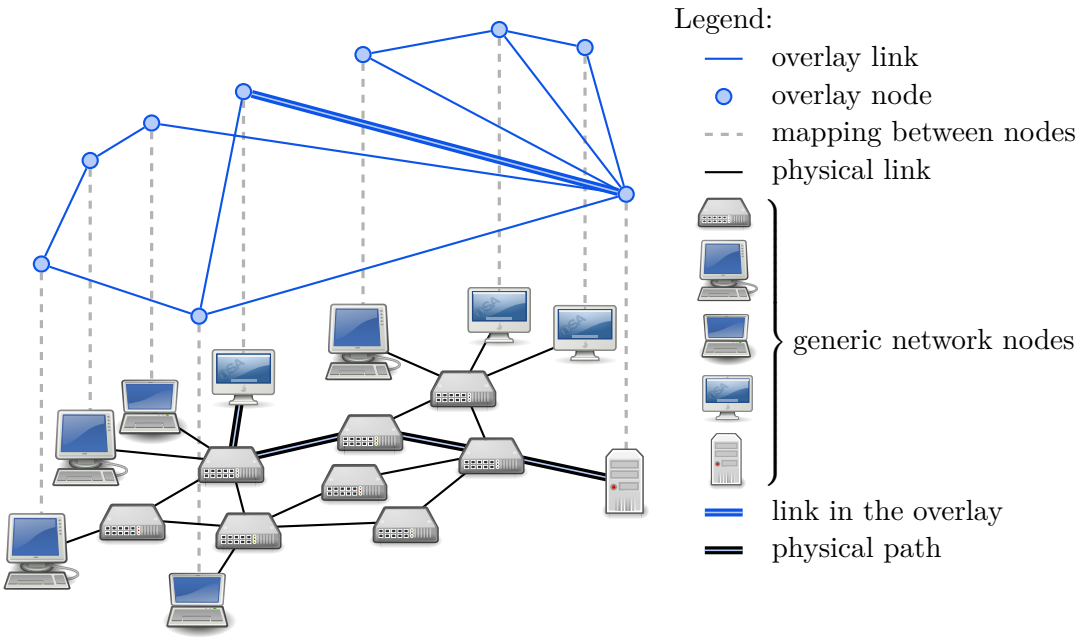


Figure 3.3: Example overlay network defined on a set of interconnected devices.

Salesforce.com's Force.com [139] supports development in a proprietary Java-like programming language — Apex; and Microsoft Azure [106] is .NET oriented.

3.1.4.3 Software as a Service (SaaS)

Finally, the top SaaS layer refers to browser-based usage of application software serving thousands of customers [77]. The underlying platforms and infrastructures are hidden from the user. There is a vast choice of such services for business and private use: Google Gmail [68] — contacts and e-mail management, Google Docs — office tools, Customer Relationship Management (CRM) from Salesforce.com [139], Facebook [52] — social networking, Flickr [55] — photo sharing; and many more.

3.2 Overlay networks

Computer networks consist of various network nodes and links. The way these devices are connected defines a topology. On top of that, an **overlay network** may be used to specify communication patterns in the computer network. In this context, nodes correspond to their physical counterparts, where links are logical — a link in the overlay network is equal to a path (possibly consisting of many links) in the underlying topology (see Figure 3.3).

A Taxonomy of the possible approaches is proposed in the Figures 3.4. As visible, there exists two main classes of overlay networks:

1. *Centralised* — where a central node is connected to the others – see Section 3.2.1,
2. *Peer-to-Peer (P2P)* – see Section 3.2.2.

The following sections offer a more detailed presentation of these classes and their respective ramifications. Also, chosen examples of the different categories are depicted in the Figure 3.5.

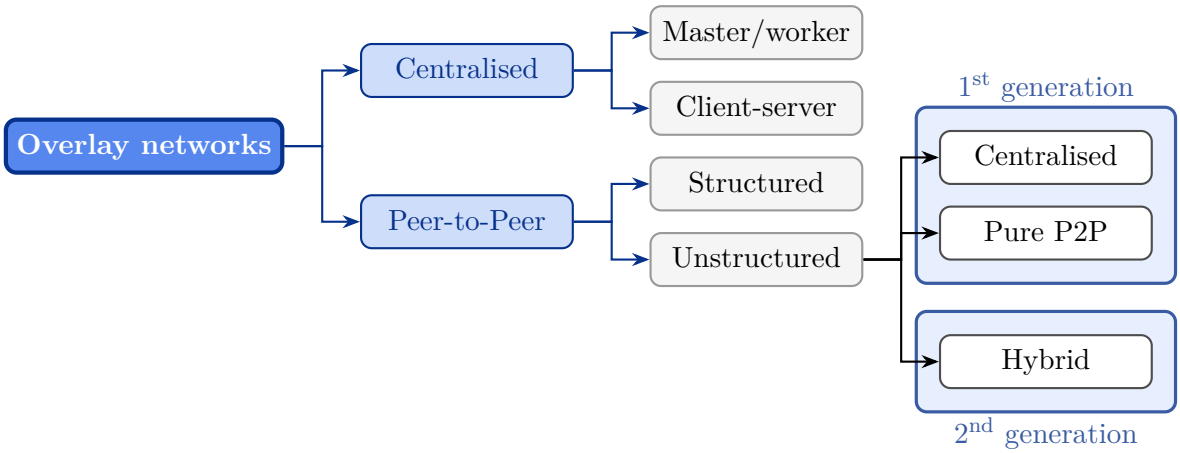


Figure 3.4: Taxonomy of overlay networks.

3.2.1 Centralised models

Centralised overlay networks have a simple structure: one node (having a central role in the system) is connected with the others. The communication in such organisation follows a request-response messaging pattern. Initiation of the message exchange is usually restricted — either the central node sends requests or it responds to them. Therefore, there are two models of centralised overlay networks: client-server and master/worker.

Many of the Internet-based applications follow the **client-server** architecture. In this context, a server provides resources or services which may be requested by one or many clients. The server waits for incoming requests. Every received request is processed and replied back to the client.

In the **master/worker** model this relation is reversed. A master node sends requests for resources or services to worker nodes. Each worker waits for incoming requests. Every received request is processed and replied back to the master node.

It is not difficult to imagine, that the central node forms a bottleneck (especially in the client-server architecture). Therefore, a device having this role must be carefully selected. Its configuration (computing power, memory, storage and network bandwidth) must match the requirements of the expected workload. In some cases, load balancing and failover systems are introduced to improve scalability of the solution.

3.2.2 Peer-to-Peer (P2P) models

Peer-to-Peer (P2P) overlay networks were created to address scalability problems for large, distributed applications. Contrary to the previously described models, the networks presented here usually consist of equally privileged, equipotent participants, called **peers**. In these schemes, symmetry in roles is present — a client may also be a server.

The **global view** (or simply the **view**) is the list of all the network members. Each participant maintains a list of some other peers representing its (partial) knowledge of the global membership, called a **partial view**. Maintenance of the global view by each peer is unrealistic in a large-scale dynamic system as it introduces considerable synchronization costs with each entering or leaving node [83]. There are two main subtypes of the decentralised overlay networks — structured and unstructured — determined by the construction of the partial view and information handling.

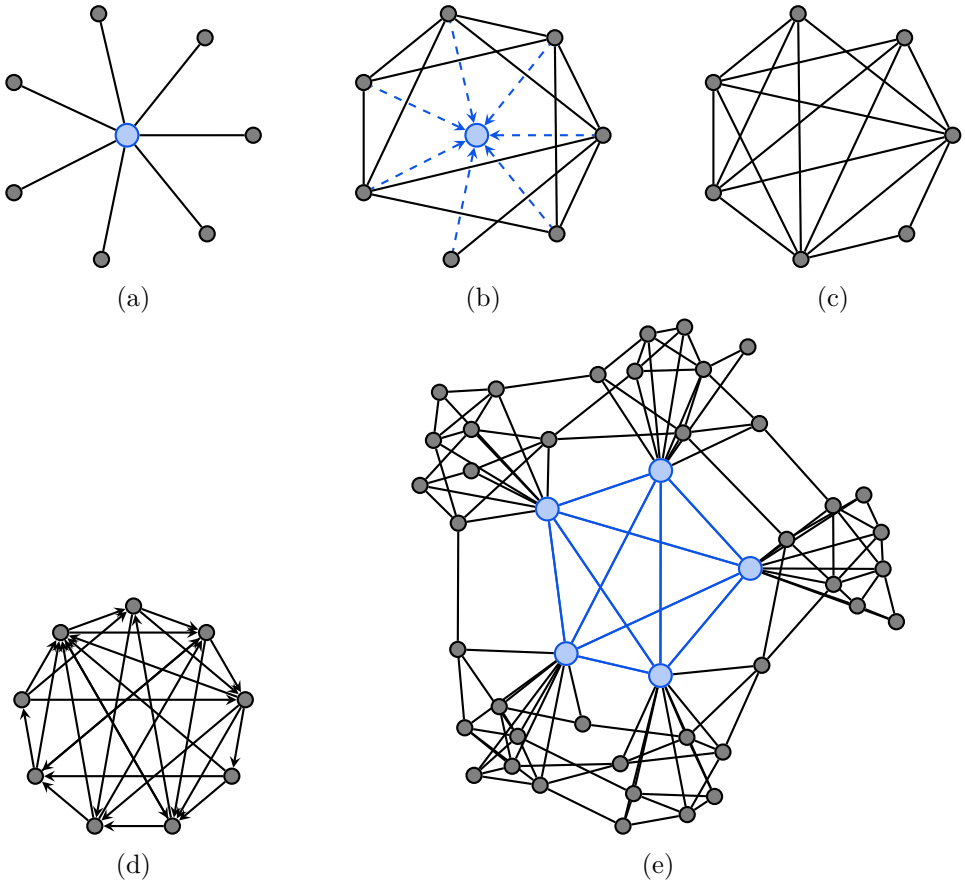


Figure 3.5: Examples of different overlay networks: (a) centralised — client-server or master/worker; (b) unstructured, centralised P2P; (c) unstructured, pure P2P; (d) structured P2P (Chord); (e) unstructured, hybrid P2P.

3.2.2.1 Structured P2P

In case of **structured P2P overlay networks**, peers are linked in a tightly controlled and organised way. The name mainly refers to approaches based on the Distributed Hash Tables (DHTs) used for distributed indexation of content represented by $(key, value)$ pairs. The structured graph enables efficient discovery (lookup) of data items ($value$) using their keys [100], where each peer within the network receives a portion of the index and a routing table to the subset of the nodes. However, that this class of systems does not support complex queries [100]. There are many protocols implementing this scheme, for example: Chord [152], Tapestry [182], Pastry [134], Content Addressable Network (CAN) [133], Kademlia [103] and Viceroy [102].

3.2.2.2 Unstructured P2P

In contrast to well-structured overlays created by the previous group of models, **unstructured P2P overlay networks** (or **random overlay networks** [162]) keep a high degree of randomness in the partial view of each peer. The first generation of models include **centralised** and **pure** P2P networks, the second — **hybrid** (or **semi-centralised**) networks [47].

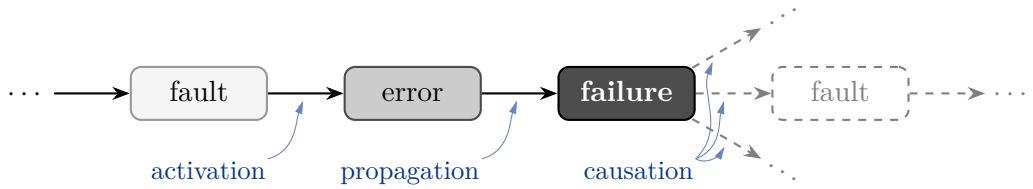


Figure 3.6: Relation between faults, errors and failures.

The idea of P2P file sharing was pioneered by Napster [118] in 1999 by the introduction of the centralised P2P network. In this approach, a central entity is necessary to provide an index of peers and the resources shared by them. This centralisation forms a possible bottleneck for massive scalability. Furthermore, it creates a single point of failure. Those problems were clearly visible when in 2001 Napster had to close its service after the lawsuit filed by Recording Industry Association of America (RIAA).

Pure P2P networks, such as Gnutella (version 0.4 [155]), Freenet [31] or Newscast [82], were developed to address the problems associated with centralisation. Without the central index, queries for content shared by the nodes are made by flooding mechanisms [100] (random walks, expanding-ring Time-to-Live (TTL) searches, gossiping, epidemic-lookup, etc.). The topology of the emerging (random) graph is self-organised — decisions on the routing are taken locally at each node. Additionally, the diameter of the network is small. All that, together with the small-world relationship between peers, is the basis of success of the mentioned protocols.

Request flooding employed in the pure P2P networks generates a potentially huge amount of signalling traffic [47]. In case of Gnutella this caused the system collapse in 2001, after enormous number of former Napster users migrated to this network within a few days [151]. To counter this problem, hybrid schemes (like Gnutella 0.6 [156]) were developed. As a result, a sort of centralisation was re-introduced into the network, creating a hierarchy between the peers. The central role is shared between designated nodes, called **Super-peers**. These **hubs** have a higher degree of connections than the rest of the nodes (**leaves**). This does not mean, that the scalability of the solution is reduced, as the amount of Super-peers scales according to the network size.

3.3 Faults, fault tolerance and robustness

There are many sources of **faults** in distributed computing [11] and they are inevitable due to the defects introduced into the system at the stages of its design, construction or through its exploitation (e.g. software bugs, hardware faults, problems with data transfer). A fault may occur by a deviation of a system from the required operation leading to an **error** (for instance a software bug becomes apparent after a subroutine call). This transition is called a fault activation, i.e. a **dormant** fault (not producing any errors) becomes **active**. An error is **detected** if its presence is indicated by a message or a signal, whereas not detected, present errors are called **latent**. Errors in the system may cause a (service) **failure** (see Figure 3.6) and depending on its type, successive faults and errors may be introduced (**error/failure propagation**).

The distinction between faults, errors and failures is important because these terms create boundaries allowing analysis and coping with different threats. In essence, faults are the cause of errors (reflected in the state) which without proper handling may lead to failures (wrong and unexpected outcome). Following these definitions, **fault tolerance** is an ability of a system to behave in a well-defined manner once an error occurs.

3.3.1 Taxonomies of faults and failures

Faults can be described using eight categories: dimension, phase of creation or occurrence, persistence, system boundaries, objective, intent, phenomenological cause, and capability [11] (see Figure 3.7 for detailed description). Each fault may belong to more than one class. Software bugs, for instance, belong to a category of software faults introduced during the development phase. Furthermore, they are permanent (until they are removed) and internal. Such faults are usually non-deliberate and human-made, caused by mistakes (accidental) or lack of professional competence, introduced without malicious intents (non-malicious).

Failures can also be characterised according to their domain, detectability, consistency and consequences [11] (see Figure 3.8). Although these classes and their subtypes are relatively general, there are specific fault models relevant in distributed computing, namely: **crash**, **omission**, **duplication**, **timing**, and **byzantine failures** [154].

The **crash** failure occurs in four variants, each additionally associated with its persistence. Transient crash failures correspond to the service restart: amnesia-crash (the system is restored to a predefined initial state, independent on the previous inputs), partial-amnesia-crash (a part of the system stays in the state before the crash, where the rest is reset to the initial conditions), and pause-crash (the system is restored to the state it had before the crash). Halt-crash is a permanent failure encountered when the system or the service is not restarted and remains unresponsive.

Omission and **duplication** failures are linked with problems in communication. Send-omission corresponds to a situation, when a message is not sent; receive-omission — when a message is not received. Duplication failures occur in the opposite situation — a message is sent or received more than once.

Timing failures are defined exactly as on Figure 3.8 and occur when time constraints concerning the service execution or data delivery are not met. This type is not limited to delays only, since too early delivery of a service may also be undesirable.

The last model — **byzantine** failure (also called **arbitrary**) — covers any (very often unexpected and inconsistent) responses of a service or a system at arbitrary times. In this case, failures may emerge periodically with varying results, scope, effects, etc. This is the most general and serious type of failure [154].

3.3.2 Dependable and secure computing

Faults, errors and failures are **threats** to system's **dependability** and **security** [11]. A system is described as dependable, when it is able to fulfil a contract for the delivery of its services avoiding frequent downtimes caused by failures. This measure has five **attributes** [11]:

- **availability** — readiness for correct service,
- **reliability** — continuity of correct service,
- **safety** — absence of catastrophic consequences on the user(s) and the environment,
- **integrity** — absence of improper system alterations,
- **maintainability** — ability to undergo modifications and repairs.

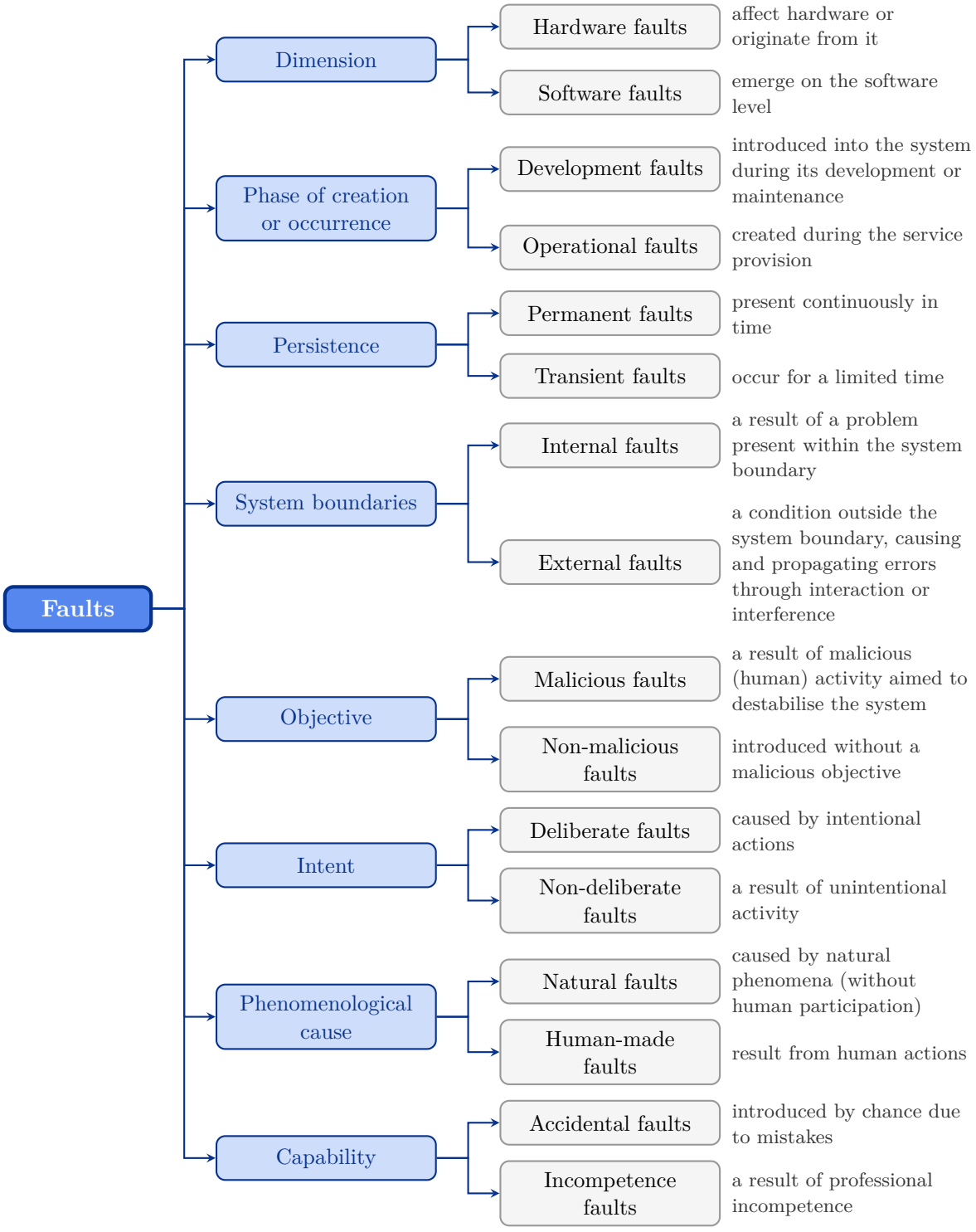


Figure 3.7: Eight elementary classes of faults [11].

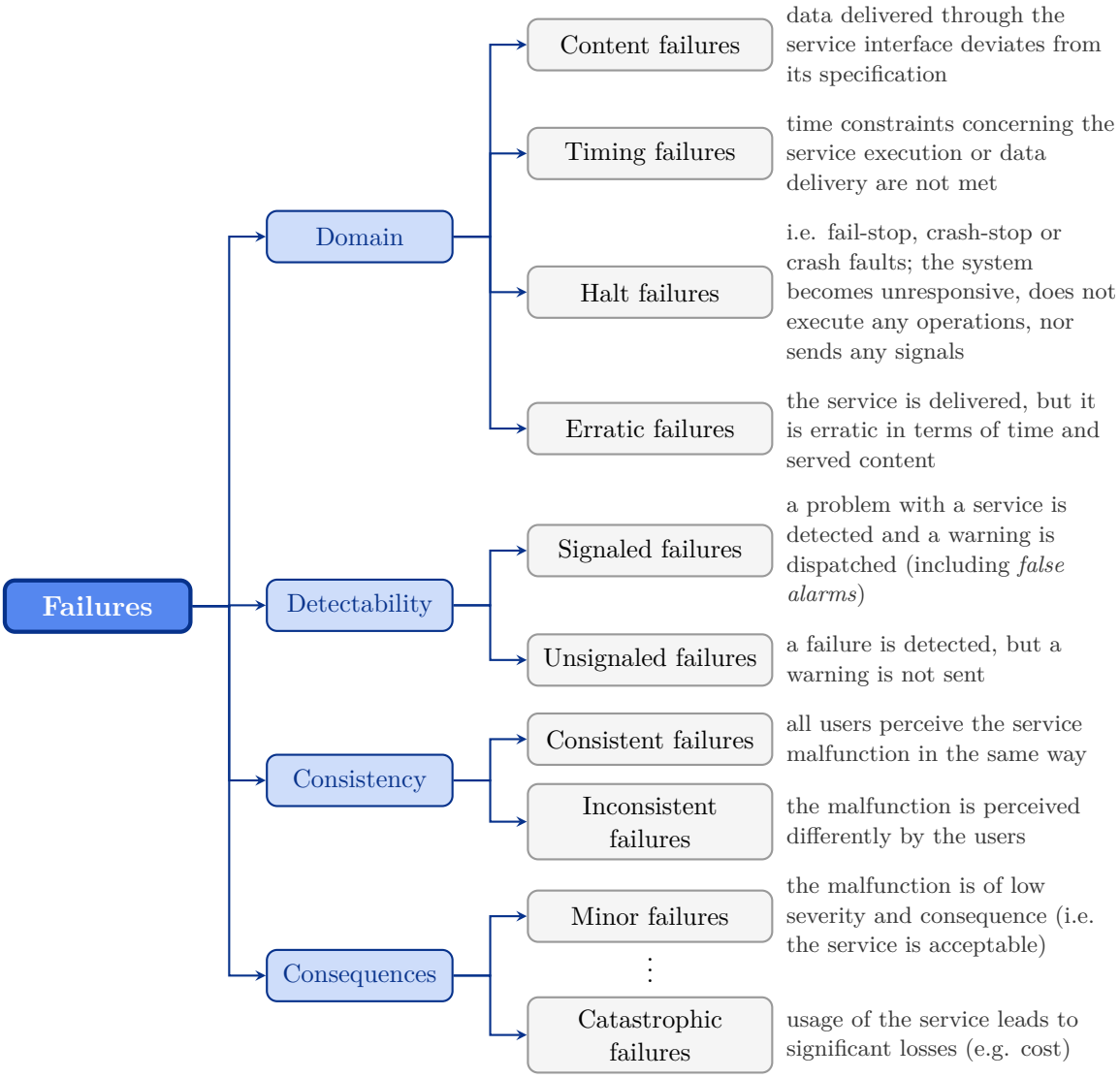


Figure 3.8: Service failures [11].

While the attributes of security are [11]:

- **confidentiality** — absence of unauthorized disclosure of information,
- **availability**
- **integrity**

} defined as previously.

Identification of threats and attributes does not automatically guarantee secure and dependable computing. For this purpose, four main groups of appropriate methods have been defined [11]: **fault prevention**, **fault tolerance**, **fault removal**, and **fault forecasting**. As visible on Figure 3.9, all of them can be analysed from two points of view — either as means of avoidance/acceptance of faults or as approaches to support/assess dependability and security. Fault tolerance techniques aim to reduce (or even eliminate) the amount of service failures in the presence of faults. The main goal of **fault prevention** methods is to minimize the number of faults occurred or introduced through usage and enforcement of various policies (concerning usage, access, development etc.) The next group — **fault removal** techniques — is concentrated around testing and verification (including formal methods). Finally, **fault forecasting** consists of means to estimate occurrences and consequences of faults (at a given time and later).

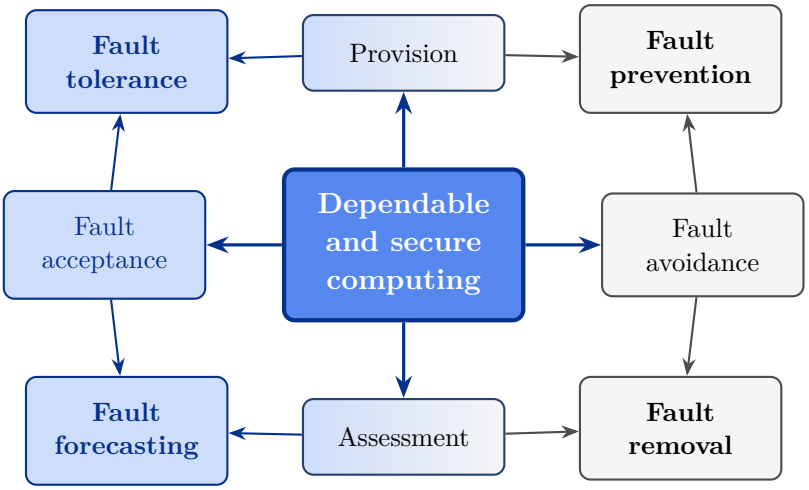


Figure 3.9: Means for dependable and secure computing.

3.3.3 Fault tolerance

Fault tolerance techniques may be divided into two main, complementary categories [11]: 1) **error detection**, 2) **recovery**. Error detection may be performed during normal service operation or while it is suspended. The first approach in this category — **concurrent detection** — is based on various tests carried out by components (software and/or hardware) involved in the particular activity or by elements specially designated for this function. For example, a component may calculate and verify checksums for the data which is processed by it. On the other hand, a firewall is a good illustration of a designated piece of hardware (or software) oriented on detection of intrusions and other malicious activities. **Preemptive detection** is associated with the maintenance and diagnostics of a system or a service. The focus in this approach is laid on identification of latent faults and dormant errors. It may be carried out at a system startup, at a service bootstrap, or during special maintenance sessions.

After an error of a fault is detected, recovery methods are applied. Depending on the problem type, **error or fault handling** techniques are used. The first group is focused on elimination of errors from the system state, while the second are designed to prevent activation of faults. In [11], the specific methods are separated from each other, where in practice this boundary is fuzzy and depends on the specific service and system types.

Generally, error handling is solved through [11]:

1. **Rollback** [49] — the system is restored to the last known, error-free state. The approach here depends on a method used to track the changes of the state. A well known technique is **checkpointing** — the state of a system is saved periodically (e.g. the snapshot of a process is stored on a disk) as a potential recovery point in the future. Obviously, this solution is not straightforward in the case of distributed systems and there are many factors to consider. In such environment, checkpointing can be coordinated or not — with differences in reliability and the cost of synchronisation of the distributed components (for details see: [34, 77, 154]).
   Rollback can be also implemented through the **message logging**. In this case, the communication between the components is tracked rather than their state. In case of an error, the system is restored by **replaying** the historical messages, allowing it

to reach global consistency [154]. Sometimes both techniques are treated as one, as usually they complement each other.

2. **Rollforward** — the current, erroneous system state is discarded and replaced with a one newly created and initialised.
3. **Compensation** — solutions based on components' **redundancy** and **replication**, sometimes referred to as **fault masking**. In the first case, additional components (usually hardware) are kept in reserve [77]. If failures or errors occur, they are used to compensate the losses. For example, a connection to the Internet of a cloud platform should be based on solutions from at least two different Internet Service Providers (ISPs).

   Replication is based on the dispersion of multiple copies of the service components. A schema with replicas used only for the purpose of fault tolerance is called a **passive (primary-backup) replication** [77]. On the other hand, **active replication** is when the replicas participate in providing the service, leading to increased performance and applicability of load balancing techniques [77]. Coherence is the major challenge here, and various approaches are used to support it. For instance, read-write protocols are crucial in active replication, as all replicas have to have the same state. Another worth to note example is clearly visible in volunteer-based platforms. An appropriate selection policy of the correct service response is needed when replicas return different answers, i.e. a method to reach quorum consensus is required [77].

These techniques are not exclusive and can be used together. If the system can not be restored to a correct state thanks to the compensation, rollback may be attempted. If this fails, then rollforward may be used.

The above methods may be referred to as **general-purpose** techniques. These solutions are relatively generic, which aids their implementation for almost any distributed computation. It is also possible to delegate responsibility for fault tolerance to the service (or application) itself, allowing tailoring the solution for specific needs — therefore forming an **application-specific** approach. A perfect example in this context is ABFT, originally applied to distributed matrix operations [30], where original matrices are extended with checksums before being scattered among the processing resources. This allows detection, location and correction of certain miscalculations, creating a **disk-less checkpointing** method. Similarly, in certain cases it is possible to continue the computation or the service operation despite the occurring errors. For instance, unavailable resource resulting from a crash-stop failure can be excluded from further use. In this work, the idea will be further analysed and extended to the context of byzantine errors and the nature-inspired distributed algorithms.

Fault handling techniques are applied after the system is restored to an error-free state (using the methods described above). As the aim now is to prevent future activation of detected faults, four subgroups according to the intention of the operation may be created. These are [11]:

1. **Diagnosis** — the error(s) are identified and their source(s) are located.
2. **Isolation** — faulty components are logically or physically separated and excluded from the service.
3. **Reconfiguration** — if redundant components are available then the previously removed are substituted with them; otherwise the service/platform is reconfigured to bypass the faulty elements.
4. **Reinitialization** — the configuration of the system (including its state) is adapted to the new conditions (removed, added or updated components).

3.3.4 Robustness

When a given system is resilient to a given type of fault, one generally claims that this system is **robust**. Yet defining rigorously robustness is not an easy task and many contributions come with their own interpretation of what robustness is. Actually, there exists a systematic framework that permits to define a robust system unambiguously. In fact, this should be probably applied to any system or approach claiming to propose a fault-tolerance mechanism. This framework, formalized in [172], answers the following three questions:

Q1	What behavior of the system makes it robust?
Q2	What uncertainties is the system robust against?
Q3	Quantitatively, exactly how robust is the system?

The first question is generally linked to the technique or the algorithm applied. The second question explicitly lists the type of faults or disturbing elements targeted by the system. Answering this question is critical to delimit the application range of the designed system and to avoid counter examples selected in a context not addressed by the robust mechanism. The third and the last question is probably the most difficult to answer, and at the same time the most vital to characterize the limits of the system. Indeed, there is nearly always a threshold on the error/fault rate above which the proposed infrastructure fails to remain robust and breaks (in some sense). This framework will be applied throughout the thesis to evaluate the effectiveness of the proposed solutions.

3.4 Summary

There are four main categories of systems for distributed computing: clusters, (cluster-based) grids, Desktop Grids and Volunteer Computing Systems (DGVCS's) and clouds. Each of them with different properties of available resources.

Overlay networks determine how tasks forming a distributed computation interact with each other. There are two centralised schemes and the distinction between them is determined by the party initiating the communication. When the interaction starts from the central node, then the execution is organised in a master/worker pattern. On the other hand, if the leaf nodes are the active side, then the scheme being used is the client-server.

In decentralised approaches, if the connections between the nodes are determined, then the organisation is structured. If however, the interactions are defined by a probabilistic scheme, then the network is unstructured.

Finally, faults, errors and failures were defined with the relations between them, their sources and types. This was followed by the introduction to the fault tolerance — the way in which such problems and the different approaches to handle failures may be handled. The chapter was closed by the definition of robustness of a distributed system — a method to quantify the fault tolerance.

Chapter 4

Evolutionary Algorithms

Contents

Evolutionary Algorithms (EAs) are a class of solving techniques based on the Darwinian theory of evolution [36] which involves the search of a **population** of solutions. Members of the population are feasible solutions and called **individuals**. Each iteration of an EA

Algorithm 4.1: General execution scheme of an EA.

```
t ← 0;
X_t ← Generate(); // generate the initial population
Evaluate(X_t); // evaluate population
while stopping criteria not satisfied do
    X̂_t ← SelectParents(X_t); // select parents
    X'_t ← Modify(X̂_t); // recombination + mutation
    Evaluate(X'_t); // evaluate offspring
    X_{t+1} ← SelectSurvivors(X_t, X'_t); // select survivors for the next generation
    t ← t + 1;
```

involves a competitive selection that weeds out poor solutions through the **evaluation** of a fitness function that indicates the quality of the individual as a solution to the problem. The evolutionary process involves at each generation a set of stochastic operators that are applied on the individuals as variations, typically recombination (or cross-over) and mutation.

The chapter starts with a presentation of a general scheme of an EA. This is followed by a detailed description of different components building the algorithm. Then, possible execution models for EAs are discussed (namely: parallel, decentralised and distributed) depending on the population structure being used. Lastly, an overview of popular, theoretical analysis approaches are presented.

4.1 General scheme of an EA

There exists many useful models of EAs yet a pseudo-code of a general scheme is provided in Algorithm 4.1. The execution starts with a creation (`Generate`) of an **initial population** (X_0) of individuals, followed by their evaluation (`Evaluate`— an assessment of quality).

The rest of the algorithm is organised in iterations, which in the context of EAs are called **generations** (indexed with a discreet time $t \in \mathbb{N}_0$). At each generation, an **offspring population** (X'_t) is created in three steps:

1. A **population of parents** ($\hat{X}_t$) is selected (`SelectParents`).
2. New individuals (X'_t) are created using variation operators (`Modify`) applied to the parents population ($\hat{X}_t$).
3. The new set of feasible solutions (X'_t) is evaluated (`Evaluate`).

The iteration ends with a selection of individuals (**survivors**) (from both sets X_t and X'_t) forming a new population (X_{t+1}) at the next generation.

The execution is terminated after a **stopping criteria** is met, e.g. a certain generation is reached.

4.2 Components of EAs

In order to define a particular EA, a number of certain components and procedures have to be specified: representation of individuals, evaluation method, population model, initialisation, parents selection mechanism, variation operators, survivors selection mechanism, and stopping criteria. The performance of the algorithm depends on the selected settings. One of its key factors is the balance between the "**exploration** of the new areas of the search

space and **exploitation** of good solutions" [2]. This may be measured with the **diversity** of the population — the number of different fitness values or individuals present — over time.

A variety of computational methods fits in the framework presented in this section: Genetic Algorithms (GAs), Evolutionary Strategies (ES's), Genetic Programming (GP) and Evolutionary Programming (EP). Details for each component are given below.

4.2.1 Representation of individuals

Representation (i.e. definition) of individuals specifies a mapping between the original problem space and an EA's **solution space** ($\mathcal{M} \neq \varnothing$). Each point in S corresponds to a feasible (i.e candidate) solution. The process of translation of the original problem space to the context of an EA is called **encoding**, and in reverse — **decoding**. There are many useful representations of individuals, these include: bit-strings $\{0,1\}^n$, vectors of real values $(x_1, \ldots, x_n)$ where $x_i \in \mathbb{R}$, trees, graphs, etc.

4.2.2 Evaluation method

The evaluation method is one of the key elements of EAs — it guides the search of the algorithm, providing the quantitative information about the quality of solutions. It defines a **fitness function** (i.e. evaluation or objective function) $f : \mathcal{M} \to \mathbb{R}$ and the pair $(\mathcal{M}, f)$ — an **optimisation problem**. The algorithm may be guided in two directions:

- **Minimisation** of f, if x^* is searched such that $f(x^*) \leq f(x), \forall_{x \in \mathcal{M}}$.
- **Maximisation** of f, if x^* is searched such that $f(x^*) \geq f(x), \forall_{x \in \mathcal{M}}$.

The assumption of minimisation or maximisation does not limit the generality of the results, as there exist the following equivalence [2]:

$$\max \{f(x) \mid x \in \mathcal{M}\} \equiv \min \{-f(x) \mid x \in \mathcal{M}\} \ .$$

Therefore, until the end of this section we will focus only on the minimisation of the optimisation problem.

Let $f^* = \min \{f(x) \mid x \in \mathcal{M}\}$ be the minimum value of the fitness function for all elements from the search space $\mathcal{M}$. The goal of the optimisation is to reach the **set of optimal solutions** $\{x \in \mathcal{M} \mid f(x) = f^*\}$, i.e. to find an **optimal solution** — an individual with a minimal fitness value.

Let $d : \mathcal{M} \times \mathcal{M} \to \mathbb{R}$ be some distance measure defined on pairs of points from the search space $\mathcal{M}$, and $N_\epsilon(x) = \{y \in \mathcal{M} \mid d(x,y) < \epsilon\}$ be a **neighbourhood** of an individual $x \in \mathcal{M}$ for some distance $\epsilon \in \mathbb{R}$ ($\epsilon > 0$). Then x is a **local optimum** if $f(x) \leq f(y) \forall_{y \in N_\epsilon(x)}$, i.e. x has the lowest fitness value among its neighbours.

4.2.3 Population model

The population plays a key role in an EA — it holds feasible (tentative) solutions to an optimisation problem. Its content changes (evolves) during the execution of the algorithm thanks to the parents and survivors selection mechanisms (described later). Yet, the size (μ) — typically remains constant.

There exist two main types of the population models: **panmictic** and **spatially-structured** — see Figure 4.1. The models with a spatial structure have two variations: **island** and **cellular** — depending on the granularity of the division.

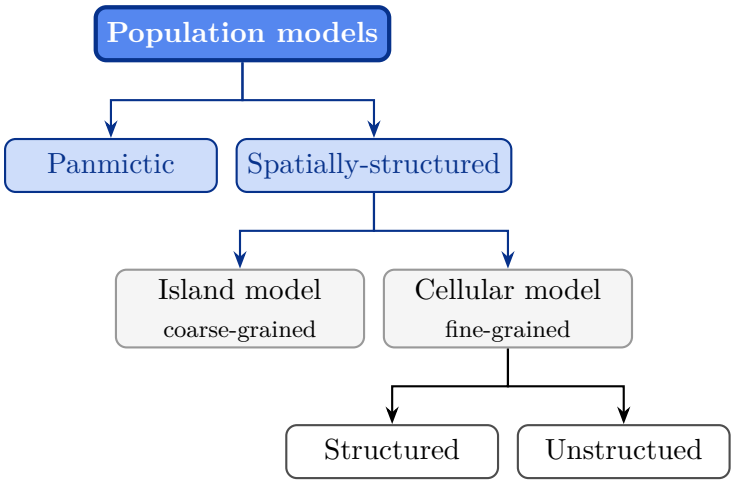


Figure 4.1: Population models in EAs — taxonomy.

Independently on the case, spatially-structured populations promote diversity in comparison to the panmictic approach [2, 26, 158]. The solutions spread slower in the population, improving the exploration properties and preventing the algorithm from getting stuck in a local optima [2].

4.2.3.1 Panmictic model

The panmictic model (see Figure 4.2a) is the simplest and most used. It is defined with a single parameter — the population size (μ, i.e. the number of individuals). To small value may cause a premature convergence, too large — leads to wasted computational resources (the waiting time for the population-wise fitness improvement might be too long) [105]. The individuals evolve freely in EAs using a single population, i.e. there are no restrictions imposed on the selection mechanisms — any individual may be chosen.

4.2.3.2 Insland model

A common coarse-grained approach is the island model [2, 26–28, 158] (see Figure 4.2b). In it, the population is split into n semi-isolated sub-populations — **islands** or **demes**, each of the size μ_i (usually equal). The parents and survivors selection mechanisms are limited to each island.

Periodically (with a frequency m_f), selected individuals — **migrants** — are sent from one sub-population (the source) to the other (the destination) according to a predefined **topology** (depicted with arrows on Figure 4.2b). This process is called **migration** and a number of migrated individuals — a **migration rate**. The choice of migrants and their integration into the island depends on the predefined migrants' selection and replacement mechanisms, e.g. the best solution in the source replaces the worst one in the destination.

The parameter settings depend on the problem being solved. Yet, general guidelines apply. Dense topologies require less function evaluations to find the global optimum [26]. Too large island sizes or too high migration rate and frequency, diminish the benefits from the spatial structure; on the other hand, too low settings decrease the quality of the solution found [26]. Finally, using too many islands is wasting computational resources and does not give any additional benefits [26].

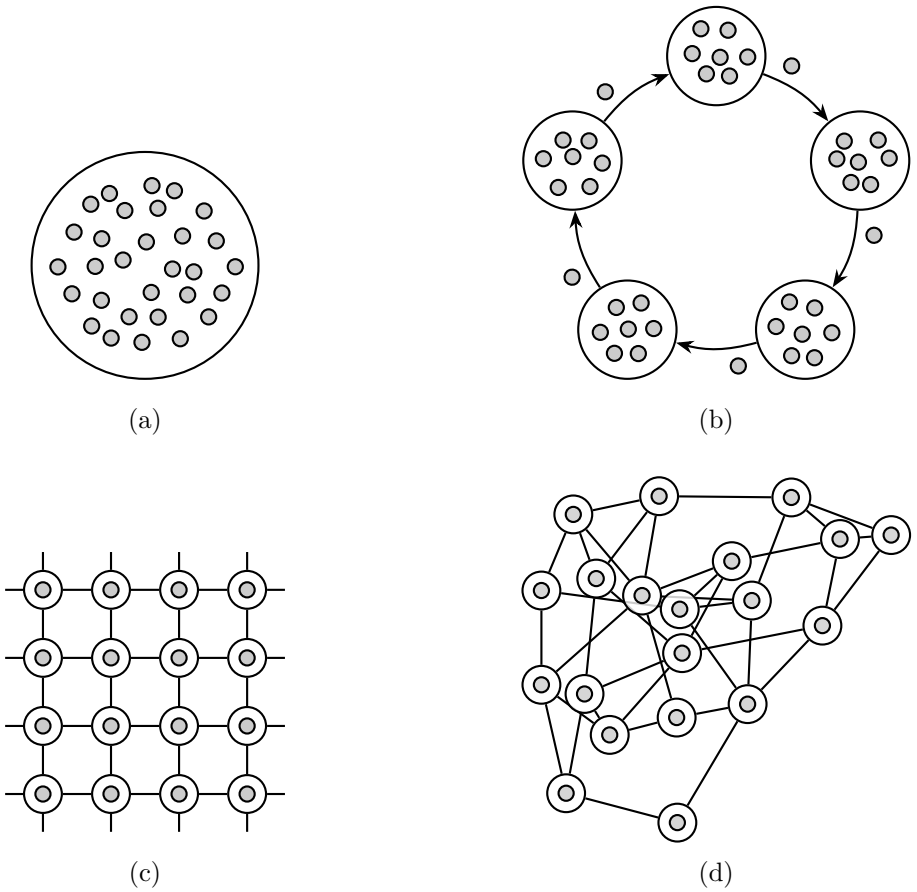


Figure 4.2: Examples of EAs population models. Each grey circle represents an individual, white circles — boundaries of the (sub-)populations. In (a) a panmictic model, (b) an island model — arrows represent the direction of individuals' migrations (with migrants above them), (c) a cellular model — structured, (d) a cellular model — unstructured.

4.2.3.3 Cellular models

In the cellular models, the population is divided into single individuals, i.e. **cells**. In the **structured** approach, the cells are organised in a regular lattice — typically in a two-dimensional grid or toroid (if the ends of the grid are wrapped around), with a predefined with (w) and height (h) (see Figure 4.2c) [2]. The parents selection mechanism is applied in a limited scope of the structure — the **neighbourhood** of the cell, which is defined with a distance on the lattice (Figure 4.3 depicts typical set-ups).

The **unstructured** cellular model is characterised by an irregular organisation of the cells (see Figure 4.2d) — in a random, small-world or scale-free graph [59, 179] (see Chapter 2). The population is defined with a size and a specific type of the network. As previously, the parents selection mechanism is limited to the neighbourhood of the cell, yet in this case — only to the immediate neighbourhood in the graph. Contrary to the structured model, the organisation of the cells does not have to remain constant throughout the execution of the algorithm, therefore the neighbourhoods may change dynamically, co-evolving with the individuals [85, 88, 179].

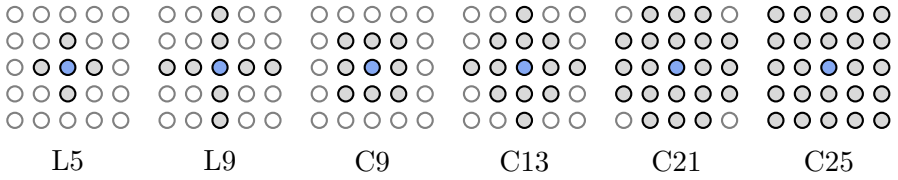


Figure 4.3: Typical neighbourhoods in a structured cellular model of an EA's population [2].

Similarly to the previously described models, the size of the population, the topology of the cells and the neighbourhood definition being used, affect the quality of the search and have to be carefully selected for a particular application [2, 59, 179].

4.2.4 Initialisation

The initialisation is a strategy for creation of the initial population. In most cases, the first generation of the individuals is created at random — the points from the search space are picked using a selected distribution. However, additional heuristics may be applied for specific problems, aiming to create an initial population with higher fitness [48].

4.2.5 Parents selection mechanism

The parents selection mechanism (i.e. mating selection) is responsible for choosing the individuals to become parents of the next generation. Better solutions should have a higher chance to be chosen, therefore the process is based on the quality of solutions (in terms of the fitness value). It is noteworthy that neglecting completely the low-quality individuals may stop the search at a local optimum, as the exploration property of the algorithm will be greatly diminished. The selected parents will be subject to mating through the application of variation operators, creating the offspring.

4.2.6 Variation operators

There are two types of variation operators, both used to create new individuals (children, offspring) based on the old ones (parents): **mutation** and **recombination** (**crossover**). Both are stochastic and problem specific. Their role is to provide the means for an EA to explore the search space. The main difference between the two types of variation operators is the number of required parameters. Mutation (slightly) alters the representation of a single individual, whereas crossover — mixes together the representations of at least two parents. The rationale behind the first operator is to increase the diversity of the population, whereas in the second case — "mating two individuals with different but desirable features, can produce an offspring that combines both of those features" [48].

Not all EA models require both types of the variation operators. For instance, in GP — mutation is often not present at all (only crossover is used), in GA — both operators are crucial, while in EP — mutation is the only variation operator used (and crossover is not present) [48].

4.2.7 Survivors selection mechanism

The survivors selection mechanism is responsible for choosing the individuals forming a population in the next generation of the EA. The choice is usually based on the quality of the solutions (their **rank**). Opposed to the previously described parents selection, the choice here is often deterministic. For instance, the best solutions from parents and offspring populations are selected (**fitness biased**) or only from the children population (**age biased**). In case when the proportion between the numbers of parents and offspring is skewed, such that the first is greater than the second — survivors selection is simply called **replacement**, as it more accurately describes the process. Typically, the size of the population remains constant, yet its alteration is also possible. The survivors selection is called **elitist** when the best individuals in the current generation are preserved for the next one at each iteration of the algorithm.

4.2.8 Stopping criteria

The stopping criteria or termination conditions determine the end of the execution. This decision depends on the problem being solved. If the optimal fitness value is known, then the stopping criteria may be based on it (optionally with a given precision $\epsilon > 0$). However, it is not always the case and what is more, there are no guarantees that a given EA will reach the optimum in reasonable time (or in a specific cases — at all). For these reasons, alternative criteria are also commonly used, like [48]: the elapsed CPU time, the number of fitness evaluations or generations, no improvement of fitness value during a given period, etc.

4.3 Parallel, decentralised and distributed executions of EAs

Execution of a simple EA requires high computational resources in the case of non-trivial problems. It might be encountered when dealing with large individuals (e.g. in case of GA — long sequences of genes, in case of GP — large parse trees) and/or large populations and/or when the objective function is CPU or memory intensive. This influences the time required to evaluate the population, which usually is the costliest operation in EAs. In such cases, time-to-solution on a single computer might be prohibitively long for practitioners (especially with usage of GP).

An example of a highly expensive EA for a computer vision problem is described in [161], where more than 24 hours is required to execute the algorithm. Another instance of even bigger requirements was reported by Melab et al. in [104] — predictive mathematical model for the concentration of sugar in beets was constructed using parallel GA, where the cumulative CPU time exceeded 27 days. Finally, the MilkyWay@home [32, 40–42] — a volunteer-based project running on the BOINC [7] platform — used to model the Milky Way galaxy represents an extreme case. It requires up to 200 hours of CPU time (with an average of 15 hours) for a single simulation and at least 30.000 simulations to optimise the input parameters [40]. This yields on average above 50 years of computation on a single CPU. Therefore, different approaches for parallel, decentralised and distributed execution of EAs were proposed to solve these issues [2, 5].

Distributed systems support three main types of execution: parallel, decentralised and distributed. The type is determined by relations and interactions between utilised resources.

		Spatially-structured	
		Coarse-grained	Fine-grained
	Panmictic	Island model	Cellular models
Parallel	✓	✓	✓
Decentralised	X	✓	X
Distributed	X	✓	✓

Table 4.1: Execution possibilities of different EA population models in distributed systems.

In a **parallel** model, one computing node (**master**) has a dominant role. It manages and supervises the execution, delegating tasks to other machines (**workers**). In a **decentralised** model, the dominant role is shared between few nodes. Each of them is responsible for managing and supervising the execution, delegating tasks to their own pool of worker machines. The master nodes may periodically exchange information. In a **distributed** model, each machine has an equal role — executing the computation semi-independently, periodically communicating with other nodes.

The schema of EAs's execution in a distributed system is conditioned on the population model being used. A summary is provided in Table 4.1 and the following sections provide the details.

4.3.1 Parallel execution of any EA

Parallel models of EAs are a well studied subject in the literature [28, 42, 66, 72]. The parallelism is usually performed at the level of individuals' evaluation, as it is the most demanding part of the algorithm. A less common option involves additionally a parallel application of the variation operators. To simplify the discussion, we will consider the parallelism only for the fitness calculations (until the end of the description of the current approach).

The computation is organised in the master/worker model (see Figure 4.4 and Section 3.2.1). Generally the panmictic (i.e. single) population (see Section 4.2.3.1) is used, following the structure of the distributed resources. The master node runs the algorithm and sends individuals to the workers for evaluation, after which — collects the results.

Two execution modes are possible: **synchronous** and **asynchronous**. In a synchronised approach, the master node waits until all individuals are evaluated before proceeding to the next generation. If the number of workers is smaller than the population size being used, then it takes more than one round of tasks' delegation to continue the evolution. Additionally, if worker nodes have different processing speeds, then the whole execution is suspended until the slowest machine returns the result [26, 158].

For a better load balancing and performance, the asynchronous mode may be used. The solution is called the **steady-state reproduction** [105] and it is characterised by the lack of the concept of generation. New individuals are continuously created (using the parents selection mechanism and variation operators) and sent for the fitness computation for each free (not computing at the moment) worker. The evaluated offspring is inserted into the population as soon as the result is returned (following the replacement mechanism).

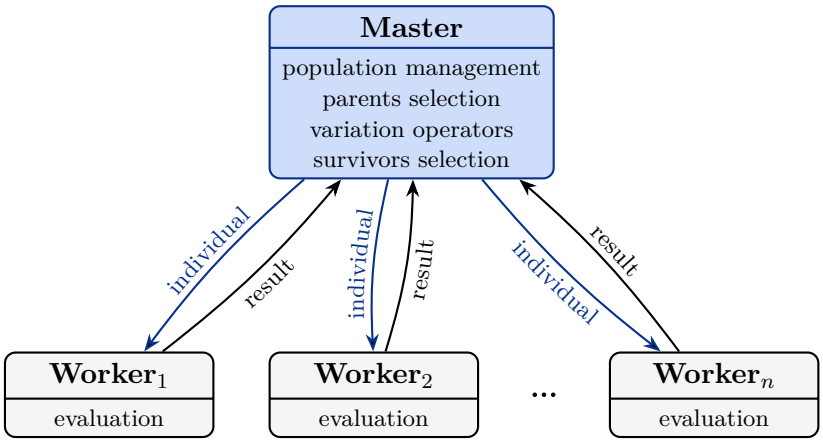


Figure 4.4: Parallel execution of an EA in a master/worker model.

4.3.2 Island-based populations: distributed and decentralised executions

Island model translates directly to distributed or decentralised executions. Each sub-population is assigned to a single computing node which administers the evolution of individuals. This forms a distributed execution. The model may be combined with a master/worker approach at the level of each island, forming a decentralised organisation of the computation (sometimes referred to as a **hierarchical parallel EA** [26, 158]).

The last important element left to define is the migration of individuals. The exchange may be performed synchronously or asynchronously, following some topology [26]. In the first case, the execution has to be stopped at predefined times (e.g. at the end of each generation) until the migrants are sent and received. In the latter — sub-populations evolve freely and the migrants are sent at convenient times. When the incoming individual is received, it may be incorporated in the population immediately or buffered to be included later. This approach achieves higher performance, as faster computing nodes do not have to wait for the slower ones.

4.3.3 Cellular-based populations: distributed executions

Cellular population models natively define the map of computing resources responsible for managing the cells, executing the EA semi-independently. Each version of the model — structured and unstructured — has their own reference implementation: the Cellular Evolutionary Algorithm (CEA) [2, 3] and the EvAg [85] (respectively). Typically, the network in the first case is static and defined by the grid structure of the population, in the second — by a self-organising, dynamic graph, created by a P2P gossip protocol: Newscast [82].

During the process of parents selection, each cell has to fetch individuals from its neighbours. However, in the case of EvAg there is an additional possibility. Some P2P protocols allow embedding application-specific data in the information exchanged between the nodes (like Newscast [82]). This option may be used to create an auxiliary neighbourhood population for each cell with a specified replacement policy. In such case, the parents selection mechanism is used on the set defined above instead of choosing from the actual neighbours in the communication graph.

Like in the previous models, there are two execution policies — synchronous and asynchronous [2]. The first version involves an auxiliary population, where the offspring is stored

until all cells are ready to replace their individuals at the end of the generation. In the asynchronous execution, the population is updated with the newly created solutions immediately (through the survivors selection mechanism). As it was studied in [2,3], the use of the latter policy allows faster convergence of the population than in the case of the synchronous one.

4.4 Theoretical analysis of EAs

There are two directions in which the theoretical analysis of EAs can be conducted, both focused on some notion of time (T). First, the study can be made to verify if the EA is able to find the solution in a finite time. This is called a **convergence analysis**, and the aim is to calculate the probability of success: $\mathrm{P}\{T \leq t\}$, for some finite t. The goal of the second approach is to estimate the **expected time** ($\mathrm{E}[T]$) required by the EA to find the optimal solution of the problem. In all cases, the literature offers theoretical results in both domains, mostly restricted to **elitist EAs**.

Note that the presentation of the foundation of probability theory or full development of the selected methods would have been too space-consuming for this manuscript. The choice of the presented mathematical tools is dictated by their popularity and this part is not aimed to act as a comprehensive overview of all approaches. The special attention has been devoted to the Markov chain model of EAs, as it will be extensively used in the Chapter 6. Therefore, the interested reader may treat this section as a starting point to the broad subject of the theoretical analysis of EAs.

4.4.1 General model of an EA used in the theoretical analysis

As EAs are "*very complex systems, involving many random factors*" [48], models and problems used in the analysis are generally greatly simplified in comparison to the algorithms executed in practice. Some variant of the $(\mu+\lambda)$ EA is typically utilised [125] and this model works as follows:

1. Randomly generate an initial population consisting of μ individuals.
2. Repeat until stopping criteria are satisfied:
   a) Create λ offspring by applying a mutation operator to λ chosen individuals from the parents population.
   b) Form a new population through a selection process from $\mu+\lambda$ individuals (parents + offspring).

For example, a typical realisation of the algorithm in the runtime analysis involves bit-string-based individuals (from a set $\{0,1\}^n$). In this case, the applied mutation operator flips each bit with a probability p. The target optimization problem tackle in this case is usually OneMax (the goal is to reach an individual consisting of maximum number of ones), LO (leading ones, where the fitness value is based on the number of ones at the beginning of an individual's bit-string representation) or other simple variant.

Despite the apparent simplicity of this model, it remains of particular interest for EAs analysis and *should not be treated as a criticism of the approach.* The theoretical work available in the field of EAs is relatively new and "*lags behind in comparison to the experimental results*" [125]. Each proof provides some new insights into the internal workings of this class of the algorithms. Even the simplest model, namely the (1 + 1)-EA, has a great importance, as emphasised in [125, 177]:

- it is very efficient for a lot of functions,

- it can not get stuck forever in a local optimum,
- it is an initial step for the analysis of more complex EAs.

4.4.2 Schema theory

The schema theory [62] proposed by Goldberg was the first, widely used mathematical tool in the analysis of EAs. It is applied on binary search spaces, where a **schema** is a hyperplane of it. This provides an aggregation method for all possible individuals. A wildcard character (`#`, may be substituted with either `1` or `0`) is used in its presentation, e.g. `11###` corresponds to all bit-strings with two ones at the beginning, which represents 2^3 instances or examples. A schema is characterised with two metrics:

- The **order** (o) — the number of wildcards in schema's presentation.
- The **defining length** (d) — the distance between the outermost defined positions (which is equal to the number of possible crossover points).

For instance, for $H = \texttt{0\#1\#\#1\#1}$, $o(H) = 4$ and $d(H) = 8 - 1 = 7$.

Let $m(H,t)$ represents the number of bit-string belonging to a schema H at a generation t, $f(H,t)$ — the observed average fitness of the schema H at the generation t and f_t — the observed average fitness of the population at the generation t, then:

$$\mathrm{E}\left[m(H,t+1)\right] \geq \frac{m(H,t)\cdot f(H,t)}{f_t}\cdot(1-p)\,. \tag{4.1}$$

Where p is the probability that the variation operators will destroy the schema H. It is defined as follows:

$$p = \frac{d(H)}{l-1}\cdot p_c + o(H)\cdot p_m \tag{4.2}$$

with l representing the length of the bit-strings, p_c and p_m — probabilities of crossover and mutation (respectively). As visible, the schemata with shorter defining length are less likely to be destroyed.

The equations 4.1 and 4.2 form the schema theorem, which yields that the "*low-order schemata with the above-average fitness increase in successive generations*" [48]. The inequality in the equation 4.1 is due to the fact that the possibility of creating a new individual from the schema through mutation is neglected. Yet, the approach is limited in applications as it does not allow explaining the dynamical or limit behaviour of EAs.

4.4.3 Markov chains and Markovian kernels

Let E be a set of a measurable space $(E,\mathcal{A})$ and T be an index set identical with $\mathbb{N}_0$ ($\mathbb{N}_0$ is chosen for convenience). Then a family of random variables $(X_t : t \in T)$ on a joint probability space $(\Omega,\mathcal{F},\mathrm{P})$ is called a **stochastic process with discrete time**. Additionally, the set E is called the **state space** of the process and set T is interpreted as points in time.
If for $0 < t_1 < t_2 < \ldots < t_k < t$ with some $k \in \mathbb{N}$ and $A \in \mathcal{A}$

$$\mathrm{P}\left\{X_t \in A \,|\, X_{t_1}, X_{t_2}, \ldots, X_{t_k}\right\} = \mathrm{P}\left\{X_t \in A \,|\, X_{t_k}\right\} \tag{4.3}$$

then $(X_t : t \geq 0)$ is called a **Markov chain**. If additionally

$$\forall_{s,t,k\in\mathbb{N}_0 \wedge s\leq t}\,\mathrm{P}\left\{X_{t+k} \in A \,|\, X_{s+k}\right\} = \mathrm{P}\left\{X_t \in A \,|\, X_s\right\} \tag{4.4}$$

then the Markov chain is termed as **homogeneous**, otherwise **inhomogeneous**.

For a homogeneous Markov chain $(X_t : t \geq 0)$ on a probability space $(\Omega, \mathcal{F}, \mathrm{P})$ with image space $(E, \mathcal{A})$, map $K : E \times \mathcal{A} \to [0, 1]$ is called a **Markovian kernel** or a **transition probability function** if

- $\forall_{A \in \mathcal{A}} K(., A)$ is measurable,
- $\forall_{x \in E} K(x, .)$ is a probability measure on $(E, \mathcal{A})$.

In particular $\forall_{x \in E} \forall_{A \in \mathcal{A}} \forall_{t \in \mathbb{N}_0} K(x, A) = \mathrm{P}\{X_{t+1} \in A \,|\, X_t = x\}$.

Let $x \in E$ be a starting state and $A \subseteq E$ be a target set, then a probability to transition from the state x to the set A within t steps is given by the t-th iteration of the Markovian kernel:

$$K^{(t)}(x, A) = \begin{cases} K(x, A) & , t = 1 \\ \int\limits_E K^{(t-1)}(y, A) K(x, \mathrm{d}y) & , t > 1 \end{cases} \tag{4.5}$$

where integration is operated with respect to an appropriate measure on $(E, \mathcal{A})$.
Finally, among the various properties of Markov kernels, for all $x \in E$ and $A \subseteq E$ the following definitions will be used throughout this chapter:

$$\int\limits_E K(x, \mathrm{d}y) \cdot \mathbb{1}_A(y) = \int\limits_A K(x, \mathrm{d}y) = K(x, A) \tag{4.6}$$

$$K(x, A) + K(x, A^c) = 1 \tag{4.7}$$

4.4.3.1 Markov model of EAs and convergence conditions

Markov chains have been used quite earlier in the literature [1] as a basis to establish the convergence in a broad sense of EAs. In this context, an EA is modeled as follows: the algorithm is executed using a population of N individuals represented by the N-tuple $(x_1, \ldots, x_N)$ with $x_i \in \mathcal{M}$ for $i = 1, \ldots, N$. $\mathcal{M}$ is a search space of a real-valued objective/fitness function $f : \mathcal{M} \to \mathbb{R}$ bounded from below (i.e. $\forall_{x \in \mathcal{M}} f(x) > -\infty$) and being minimized by the algorithm. This gives a state space $E = \mathcal{M}^N$. The population at step $t = 0$ (called the **initial population**) is created using some initial distribution $p(.)$ yielding the random population $X_0 = (X_{0,1}, \ldots, X_{0,N})$. A population $X_t = (X_{t,1}, \ldots, X_{t,N})$ at a time $t > 0$ is generated by so-called genetic operators (described by the associated stochastic kernels $K(.,.)$). These operators permit evolving the set of feasible solutions and they depend only on its state from the previous step $(t - 1)$.
The above model leads to a conclusion that the stochastic sequence $(X_t : t \geq 0)$ is a Markov chain.

To analyse the ability of an EA to converge in some sense to a specific set associated with globally optimal solutions of the optimization problem, the term **convergence** has to be precisely defined.

Definition 4.4.1 (Complete convergence [136])**.** *Let (D_t) be a sequence of random variables defined on a probability space $(\Omega, \mathcal{F}, P)$. Then (D_t) is said to* ***converge completely*** *to 0, denoted as $D_t \xrightarrow{0} 0$, if for any $\epsilon > 0$*

$$\lim_{t \to \infty} \sum_{i=1}^{t} P\{|D_i| > \epsilon\} < \infty \tag{4.8}$$

Definition 4.4.2 (Convergence in probability [136]). *Let (D_t) be a sequence of random variables defined on a probability space $(\Omega, \mathcal{F}, P)$. Then (D_t) is said to* ***converge in probability*** *to 0, denoted as $D_t \xrightarrow{\mathrm{P}} 0$, if for any $\epsilon > 0$*

$$\lim_{t\to\infty} P\{|D_t| > \epsilon\} = 0 \tag{4.9}$$

It is worth noting here that complete convergence implies convergence in probability.

For the ease of the further analysis, additional notations are defined as follows:

- $b(X_t) = min\{f(X_{t,i}) : i = 1, \ldots, N\}$ — the best objective/fitness function value of population X_t at step $t \geq 0$;
- f^* — the global minimum of the objective/fitness function $f : \mathcal{M} \to \mathbb{R}$;
- $d(X_t) = b(X_t) - f^*$ — a distance of the best objective/fitness function value of population X_t at step $t \geq 0$ to the global optimum f^*;
- $A_\epsilon = \{x \in E : d(x) < \epsilon\}$ — a set of ϵ-optimal states with $\epsilon > 0$;
- $B(x) = \{y \in E : b(y) \leq b(x)\}$ — a set of states which are better than or equal to a state x according to the objective function.

At this point it is possible to introduce the main theorem on top of which the analysis conducted here is based.

Theorem 4.4.1 (Condition of EA convergence [136]). *Let $A_\epsilon = \{x \in E : d(x) < \epsilon\}$ with some $\epsilon > 0$ be the set of ϵ-optimal states. An evolutionary algorithm, whose stochastic kernel satisfies*

- $\forall_{x \in A_\epsilon^c = E \setminus A_\epsilon} K(x, A_\epsilon) \geq \delta > 0$
- $\forall_{x \in A_\epsilon} K(x, A_\epsilon) = 1$

will converge to the global minimum of a real-valued objective function $f : \mathcal{M} \to \mathbb{R}$ with $f > -\infty$ defined on an arbitrary space $\mathcal{M}$, regardless of the initial distribution.

Proof. See [136]. □

Corollary 4.4.1. *The random sequence $(d(X_t) : t \geq 0)$ converges in probability to 0 (denoted as $d(X_t) \xrightarrow{\mathrm{P}} 0$) and converges completely to 0 (denoted as $d(X_t) \xrightarrow{0} 0$).*

Now that we have reviewed the mathematical results that permit to define the conditions of EAs convergence, we are able to analyse the ability of an EA to converge or not to "good" solutions. Such an analysis is proposed in the Section 6.1. More generally, the results presented in the Chapter 6 inherits and adapts the Markov model of EAs to take into account the cheating resilience.

4.4.4 Artificial fitness levels

The approach was introduced in the runtime analysis of (1+1)-EA, dealing with pseudo-boolean fitness functions (see Definition 4.4.3). Later, the method was extended for usage in more general settings [125].

Definition 4.4.3 (Pseudo-boolean function [178]). *A pseudo-boolean function $f : \{0,1\}^n \to \mathbb{R}$ is a degree-k function with N non-vanishing terms if it can be represented as*

$$f(x_1, \ldots, x_n) = \sum_{1 \leq i \leq N} w_i \prod_{j \in S_i} x_j \tag{4.10}$$

where $w_i \in \mathbb{R} - \{0\}$ and the size of the sets $S_i \subseteq \{1, \ldots, n\}$ is bounded above by k. Degree-1 functions are called linear and degree-2 functions are called quadratic.

Artificial fitness levels are based on creating a partition of the search space ($\mathcal{M}$) into sets $A_1, \ldots, A_m$ (where $m < |\mathcal{M}|$) called *fitness levels* that are ordered w.r.t. fitness values (i.e. $A_1 <_f A_2 <_f \cdots <_f A_m$, see Definition 4.4.4). We say that the algorithm is in A_i or on level i if the current individual is in A_i.

Definition 4.4.4 ($<_f$-partition)**.** *For $A, B \subseteq \{0,1\}^n$ and $f : \{0,1\}^n \to \mathbb{R}$ the relation $A <_f B$ holds if $f(a) < f(b)$ for all $a \in A$ and $b \in B$. An $<_f$-partition of $\{0,1\}^n$ into non-empty sets $A_1, \ldots, A_m$ is when $A_1 <_f A_2 <_f \cdots <_f A_m$ and all $a \in A_m$ are global optima.*

The introduced partition allows creating a Markov chain using only the m different states. This leads to a smaller Markov chain transition matrix and simplified calculations [125].

Yet, the fitness value of the population in the elitist EAs can never decrease. Therefore, if one can derive lower bounds on the probability of leaving a specific fitness level towards higher levels, this yields an upper bound on the expected running time (see Theorem 4.4.2).

Theorem 4.4.2 (Fitness-level method [98])**.** *For two sets $A, B \subseteq \{0,1\}^n$ and a fitness function f let $A <_f B$ if $f(a) < f(b)$ for all $a \in A$ and all $b \in B$. Partition the search space into non-empty sets $A_1, A_2, \ldots, A_m$ such that $A_1 <_f A_2 <_f \cdots <_f A_m$ and A_m contains global optima. For an elitist EA let s_i be a lower bound on the probability of creating a new offspring in $A_{i+1} \cup \cdots \cup A_m$, provided the population contains a search point in A_i. Then the expected number of iterations of the algorithm to find the optimum is bounded by*

$$\sum_{i=1}^{m-1} \frac{1}{s_i} \,. \tag{4.11}$$

We used the extension of the Theorem 4.4.2 in [112] to establish the expected runtime of an elitist, parallel (1+1) EA with cross-over built on top of a Small-World network. Since this study does not take into account cheating faults, it has not been integrated as a core contribution in this manuscript, yet the corresponding article have been provided in Appendix B as an example of a full runtime analysis.

4.4.5 Dynamical systems

Dynamical systems is a method proposed by Vose [170]. The approach is based on an infinite population. Instead of analysing single individuals, the set of candidate solutions is described as a n-dimensional vector $\vec{p}$, where n is the size of the search space. Each entry p_i^t of the vector contains a proportion of individuals of a type i at a time t.

For the definition of the model, two matrices are needed: M and F — mixing and selection matrices (respectively). The first one describes the effects of variation operators (mutation and crossover), the second one — the effects of selection on each individual for a given fitness function. Based on that, a "genetic operator" is defined as a product of both matrices: $G = F \circ M$.

G is a trajectory of the population evolving on a n-dimensional surface. The next set of candidate solutions is generated through the application of G to the current population: $\vec{p}^{\,t+1} = G\vec{p}^{\,t}$. The "genetic operator" matrix allows identifying attractors in the search space

— the points towards which the population is drawn. For a full analysis see [171] where the method is applied to a GA with a fitness proportional selection, a one-point crossover and a bitwise mutation.

4.4.6 Interacting Particle Systems (IPS)

Interacting Particle Systems (IPSs) are used to model and study the asymptotic behaviour of GAs [38, 39]. The approach is based on the Markov chains framework to describe the population and its transitions during the evolutionary process. The GA is defined as a system of particles from a given measurable space $(E, \mathcal{A})$, in which it randomly evolves driven by a given fitness function. Its population of $N \geq 1$ individuals is modelled as a Markov chain (not necessarily time homogeneous, see Section 4.4.3): $X_t = (X_t^1, \ldots, X_t^N)$ for $t \geq 0$, where $X_t^i \in E$. Optionally, the initial particle system X_0 consists of N independent particles with common law $\eta_0 \in \mathcal{P}(E)$, where $\mathcal{P}(E)$ is a set of all probability measures with the weak topology on E.

The flow of the empirical measures m associated with the systems of particles X_t is used instead of "*studying the dynamic and limiting process of the so defined Markov chains*" [39]:

$$m(X_t) \stackrel{\text{def}}{=} \frac{1}{N} \sum_{i=1}^{N} \delta_{X_t^i} \tag{4.12}$$

where δ_x is a Dirac measure at $x \in E$.

The empirical measure $m(X_{t-1})$ of the population at the time $t-1$ determines the transition to the next one (X_t). Specifically, the next generation X_t consists of N (conditionally) independent random variables with a probability measure $\Phi_t(m(X_{t-1}))$, where $\Phi_t : \mathcal{P}(E) \to \mathcal{P}(E)$, $t \geq 1$, is a given collection of sufficiently regular functions defined by the GA operators and parameters. Following [39]:

> *In some sense to be defined the empirical measures* $\eta_t^N \stackrel{\text{def}}{=} m(X_t)$, $t \geq 0$, *converge as the number of particles* $N \to \infty$ *to a deterministic flow of distributions* $\eta_t \in \mathcal{P}(E)$, $t \geq 0$, *solutions of the measure valued dynamical system* $\eta_t = \Phi_t(\eta_{t-1})$, $t \geq 1$. *In the measure valued processes literature this system is usually called the limiting process.*

Therefore, the solution forms an infinite population model using a distribution measure on the search space to describe the behaviour of the finite population [39].

4.4.7 Drift analysis

Drift analysis is a very powerful theoretical tool available for estimating the expected runtime of EAs. It remains useful when the EA has non-monotonic progress, i.e. when the fitness value is a poor indicator of progress. Informally, it shows how the challenging problem of predicting the long-term behaviour of a given evolutionary meta-heuristic can be reduced to the often trivial problem of describing how the state of the heuristic changes during one iteration.

This approach derives from the theory of *Martingales*, a notion used in the nineties to prove the convergence of EAs using non-elitist selection strategies [137]. The following description is directly inspired by [125] and the seminal work of He & Hao [73].

Let S^* be the set of populations containing the optimal solution of an optimization problem, S the set of all possible populations, and $d(X_t)$ a distance function for measuring the distance from population X to S^*, at time-step t. The drift of the random sequence $\{d(X_t), t = 0, 1, \ldots\}$ at time t is defined by

$$\Delta\left(d(X_t)\right) = d(X_t) - d(X_{t+1})$$

Let's define the stopping time of an EA as $\tau = \min\{t : d(X_t) = 0\}$, which is the first hitting time on the optimal solution. The main motivation in the concept of drift analysis is that it may often be easier to estimate the drift of a random sequence rather than τ directly from the Markov chain. Thus the idea behind drift analysis is quite straightforward:

> *If the distance of the current solution from the optimal one is d and if the drift towards the optimal solution is greater than Δ at each time step, we would need at most $\frac{d}{\Delta}$ time steps to find the optimal solution. Hence the key issue here is to estimate Δ and d.*

In [73, 74], drift conditions are given to determine whether the average time complexity of an EA to solve a given problem is polynomial or exponential in the problem size. Numerous drift theorems are available (Additive, multiplicative, variable, population drift etc.) and actually, a significant number of them originate from other fields than EC. In general, drift analysis lead to simpler runtime proofs, assuming a good distance function is defined.

4.4.8 Reductionist approaches

Reductionist approaches have their roots in physics and engineering, where the parts of a complicated system are examined in separation from each other (interactions are neglected). A typical decomposition of EAs involves analysis of selection-only or variation-only algorithms [48].

In the first case, the term **takeover time** was introduced [63]. It is defined as a time required for the fittest individual to completely take over the population. On top of this approach many parents and survivors selection mechanisms were analysed, using different theoretical tools such as Markov chains, order statistics or differential equations [12–14, 22, 29, 138, 150].

For variation-only algorithms, the **mixing time** was defined as a time at which the "*recombination brings together building blocks initially present in different members of a population*" [48] [157]. The outcome of the analysis was that the discussed value should be less than the takeover time to guarantee a well-performing EA. If it is not a case, then the fittest individual takes over the population, removing some potentially useful solutions without giving them the chance to be used in the evolutionary process.

4.5 Summary

In this chapter EAs were introduced. Their general execution schema was presented and all components were discussed. In some cases, the realisation of an EA might be very costly in terms of CPU and memory/storage requirements. Therefore, different models for parallel, decentralised and distributed executions were introduced. Some well-known approaches for implementation of the schema in distributed systems were discussed (based on the population

structure), including: a parallel model for any EA, the well-known Island Model, Cellular Evolutionary Algorithms (CEAs), and Evolvable Agents Model (EvAg).

The fundamental knowledge about the theoretical analysis was presented. There are basically two directions in which the study may be conducted, both focused on some notion of time (T). Either the analysis aims to determine if an EA is able to find the solution in finite time (**convergence analysis**) or to estimate the expected time required to find the optimal solution of the problem (**expected runtime analysis**). Various approaches exists for both directions. Providing an exhaustive overview would have been out of the scope of this manuscript thus we focused only a brief insight of some results being of interest in this context.

Part II

Cheating-Tolerance of Parallel and Distributed EAs in DGVCS's

Chapter 5

Fault-Tolerant Executions of EAs in DGVCS's

Contents

This chapter starts with an introduction of main challenges for executing applications in DGVCS's. The concept of resource availability in volunteer-based platforms is presented with its main characteristics and metrics. Formalisations of cheating faults and cheating-tolerance are given, where the first one is used to describe the effects of activities performed by malicious users (cheaters) and the second — the system's resilience to them. The cheater model used in this work is described in detail.

After that, a general guideline is provided helpful when deciding to use the platform for the execution of a particular case of the optimisation problem and the EA model. The choice is mainly dictated by the expected performance of the solution.

Finally, fault-tolerant aspects of EAs are presented with the stress on the crash- and cheating-failures in DGVCS's. Existing approaches to achieve a fault-tolerant execution of the algorithm are described. The chapter ends with a presentation of open problems in this area of research.

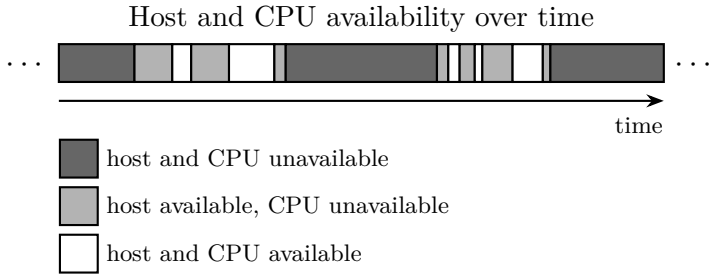


Figure 5.1: Illustration of Host and CPU availability over time.

5.1 Challenges in DGVCS's

As mentioned in Section 3.1.3, DGVCS's are very challenging platforms from developers' point of view. The main challenges are failures caused by resource volatility, compromised security and insufficient performance. As introduced in Section 3.3.1, each of them on its own may be a source of various problems. In practice, a significant number of the machines participating in the platform will commit at least a single (Input/Output (I/O)) error over time (around 35%) [90]. Moreover, the error rate is not stationary and varies in the course of time [90]. Furthermore, "*about 70% of the errors are caused by only 10% of the hosts*" [90].

Nevertheless, some errors and failures are easier to handle than the others. For example, common omission and duplication failures are usually detected and corrected at the message transport level. Timing faults are commonly caused by the problems in the design of the platform or application. Crash and byzantine failures are the most difficult to handle, therefore will be presented in this section in detail.

5.1.1 Resources availability

As mentioned above and in Section 3.1.3, a DGVCS creates a very volatile environment. The number of resources available for the computation varies over time. These fluctuations are a commons source of crash failures.
There are two ways in which a **resource availability** can be measured in DGVCS, these are: **host availability** and **CPU availability** (see Figure 5.1). The first one expresses the probability that a system is operational at a given time:

$$H_a = \frac{\text{uptime}}{\text{uptime} + \text{downtime}}$$

where the actual up- and down- times may be measured as times when the system is powered on and off (respectively). The other possibility, more applicable in the context, is to use the reachability of a device through the network.

Unfortunately, the host availability alone is not able to fully grasp the dynamic changes in the availability of resources in a DGVCS. This problem is connected with the utilisation of idle cycles for computation. To solve it, the CPU availability was introduced: the CPU is available when it is possible to run the application on the volunteered resource and it is unavailable otherwise. Its probability may be computed as:

$$CPU_a = \frac{\text{time of CPU availability}}{\text{uptime} + \text{downtime}}$$

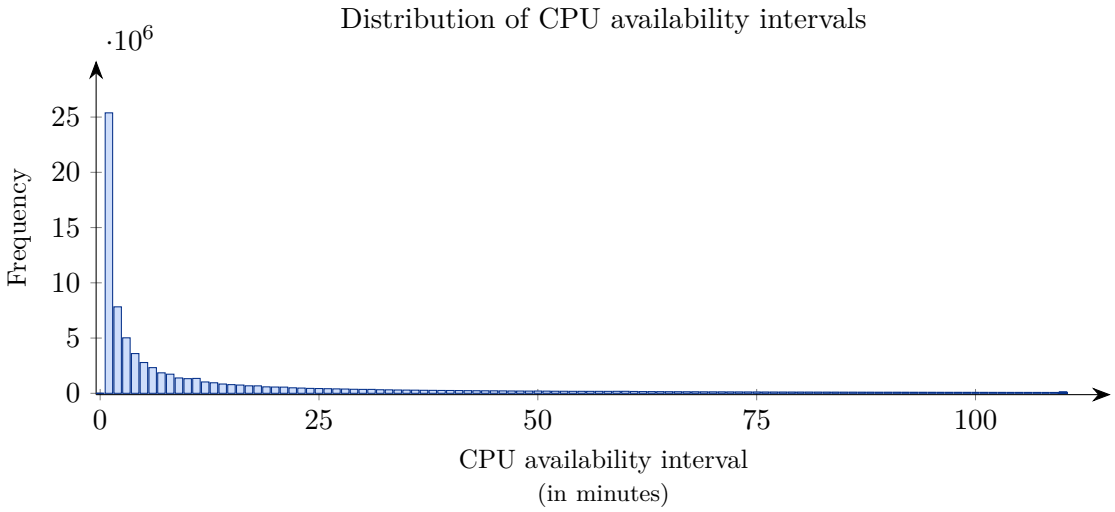


Figure 5.2: A distribution of CPU availability intervals measured for the BOINC client of the SETI@home project. The data, extracted from the Failure Trace Archive (FTA) [53], was collected between April 1st, 2007 to January 1st, 2009. It includes more than 100.000.000 CPU availability traces of almost 230.000 hosts scattered all over the world, in total capturing almost 58.000 years of CPU time. The results presented here are limited to intervals up to 110 minutes.

where uptime is the time when the client application is connected to the platform and downtime is the time when it is not connected.

There are numerous studies on the subject in the areas of DGVCS's [80], HPC Clusters [124, 142], Grids [78, 181], Domain Name System (DNS) servers [130], web servers [15], and P2P networks [20, 70, 141, 146]. All analysing traces gathered from the real projects with the purpose of building prediction models, characterising resources, and improving scheduling of jobs. A diverse collection of such data is available in the Failure Trace Archive (FTA) [53,79,92]. The datasets differ in measurements taken (host versus CPU availability), resource types (private, enterprise, home, work, school), scale (number of hosts participating) and duration (the time period over which the measurements are taken).

The distinction between the host and CPU availabilities is crucial. For example, the majority of hosts in DGVCS's and P2P networks are available for less than an hour [20, 80, 141, 146]. However, the actual time during which the computation may run on the resource in the volunteer fashion may be just a fraction of the previously mentioned value. In extreme cases, this range may consist of many, very short intervals of CPU availability (for example, each of them below one second) with the host available for 99% of time. As a result, such a resource would not be very useful (or even completely useless) for distributed computation [80].

Figure 5.2 presents distribution of CPU availability intervals captured for the BOINC client of the SETI@home project [80, 81]. The data was collected between April 1st, 2007 to January 1st, 2009. It includes **more than 100.000.000 CPU availability traces** (and the same number of unavailability traces) of **almost 230.000 hosts** scattered all over the world, capturing in total **almost 58.000 years of CPU time**. In the study of Javadi et al. [80], from which we acquired this data, the actual CPU utilisation of a host is ignored. Instead, the resource is considered as available for computation if the associated with it

BOINC client reports it as "free". The exact definition of this term is one of the client-side configuration options. Finally, as emphasised in the study, the data was recorded at the level of the BOINC client and it is independent from the SETI@home project. This means, that if the host was executing any of the project's applications, it is still represented in the trace as available.

It can be seen from the data in Figure 5.2 that the distribution of interval's lengths is significantly biased towards short CPU availability times. We can observe this trend in the data at all scales (minutes, hours, days, etc.). It is worth to note here, that only 497 intervals were shorter than 1 minute (barely visible on the figure). The maximum CPU availability time captured by the study equals to 13216 hours ($\approx$ 550 days). It is important to note here that **more than 25% of the collected intervals are equal to or shorter than one minute**. Therefore, the assertion that this platform is highly volatile is not an exaggeration.

The availability status may change for a number of reasons. In DGVCS's, the most likely cause of this may be that the resource is no longer idle (e.g. a mouse or a keyboard became active) or a failure occurred (see Section 3.3.1). If the status change is the result of a deliberate (but non-malicious), user's action (e.g. scheduled downtime, user closes the application, etc.) then the system can be notified (i.e. graceful shutdown). The dynamic changes over time of resources availability and membership in the platform is called the **churn** [153]. Such variations are common and unavoidable in DGVCS's. Therefore, the system has to be designed to handle them transparently, for instance by the reconfiguration or through the built-in self-organisation. The worst possible scenario of churn is when a given portion of the resources frequently leaves the system or crashes and is immediately replaced by the new ones [83]. It is so, because it not only requires multiple reconfigurations of the platform, but also these new resource have to be initialized.

5.1.2 Cheating faults and cheating-tolerance

Volunteers in DGVCS's are tempted to participate in the platform through various incentives. There are five basic mechanisms to provide such motivators, all with their own challenges and drawbacks [95]:

- Payment-based — participation is rewarded with a payment. It may be a real monetary reward or virtual (i.e. "*some tokens that can be redeemed for other services*" [95]).
- Auction-based — the resources are auctioned, where members of the platform are chosen through bidding (depending on the context, with the highest or the lowest price).
- Exchange-based — the service is exchanged between the participants in a synchronous and stateless transaction (pure barter-based solution).
- Reciprocity-based — contrary to the exchange-based approach, interactions involving resource sharing have a state and a history, which creates the base for the exchange.
- Reputation-based — a generalised version of the previous approach. "*The system records a score for each participant based on the assessment made by other members of the platform*" [95].

However, in scientific projects like SETI@home [9] or Folding@home [56] the incentive is usually based purely on fame. Each active member is rewarded with points proportionally to the contribution. In turn, these scores are used to create publicly available rankings of participants.

5.1.2.1 Cheater's objectives and cheating possibilities

Usually, malicious users of DGVCS's seek to obtain mentioned rewards with little or no contribution to the system. Their main objective is to stay undetected as long as possible maximising the profit. This activity may be purely **passive**, when the aim is to obtain the payment with as little work as possible. It may also be **active** with a goal to delay the end of the execution. This is motivated by the fact that the increased amount of work (therefore overall computation time) should also yield more possibilities to obtain the gratification.

However, obtaining offered incentives does not have to be the only motivation. "Crackers" or "black hat" hackers are a perfect example of such approach. These are users which "*violate computer security for little reason beyond maliciousness, personal gain or satisfaction*" [110].

Generally, malicious users can simulate or cause any failure relevant in distributed computing (see Section 3.3.1). However byzantine failures are the most useful, flexible, and hardest to detect. The rest are easily discovered and their effect is very limited (usually only to the node causing the problems).

Intentional introduction of byzantine failures into the system may be done by altering (i.e. corrupting) results or messages being sent. As such data is not true (in the meaning of not genuine or not authentic), it may be defined as **cheated**, the process of its production and transfer as **cheating**, and the user sending it as a **cheater**.

Cheaters may organise themselves into **colluding** groups with the aim to feed incorrect results [25]. It might be voluntary or not. The second case is usually the effect of a virus or a bug in the code [25].

5.1.2.2 (Lazy) Cheater Model

Distributed computation can be defined in terms of tasks, their assignment and execution on the available resources. Therefore, cheating at the level of a task (T) execution will be now formalized, following its first definition in [168].

Let $W_1(T)$ be the total work i.e. the number of unit operations of T and $W_\infty(T)$ be its depth work, i.e. the maximal number of unit operations on a critical path when executed on an unbounded number of processors. For the sake of simplicity, we assume that for each task T that takes part of the execution, there always exists a cheated version T' that secretly replace T when cheating is conducted. In the sequel, T' is defined with the following constraints:

1. The prototype of T', i.e. its signature (as defined in the C/C++ programming language) is unaffected by the cheating. In particular, T' has the same function name, arity, argument types, and (more importantly) return type than T. This is justified by the fact that the cheaters want to stay in the system as long as possible. Falsifying the return type of task will probably lead to a crash fault that can help to immediately identify the cheater. Whereas, if the attacker is satisfied with corrupting only the return value (and not its type), he can expect this modification to be hidden during a sufficient long time for him to stay in the system and collect credits.
2. The result of T' is altered such that it has the worst impact on the execution without consuming more computing power than T, i.e.

$$W_1(T') \leq W_1(T) \quad \text{and} \quad W_\infty(T') \leq W_\infty(T) \tag{5.1}$$

This last constraint models "lazy" cheaters that want little or no contribution to the system. The notion of "the worst impact" is more difficult to define and is specific to the considered application and granularity of the execution. Its role is to provide an upper bound on the cheating impact in the analysis conducted here.

Additionally, it is assumed that the cheaters possess **full knowledge** about the computation and the platform. Such assumption is not far from reality, because most of the volunteer-based projects release the sources of the applications to the public. This is done in order to gain the trust of the volunteers. However, it also increases the system's susceptibility to attacks, because anyone with the proper technical knowledge can carry out an analysis of the published materials.

5.1.2.3 Cheating-tolerance

Similarly to a definition given in Section 3.3, **cheating-tolerance** may be defined as an ability of a system to behave in a well-defined manner once cheating occurred in the system. There are few possible approaches to achieve tolerance of this kind of errors in DGVCS's. The main directions are:

1. Usage of a special algorithm for task' scheduling [25], where cheating-tolerance is achieved through redundancy (or compensation) (see Section 3.3.3).
2. Implementation of an ABFT design. In this case either specific features of the executed algorithm are exploited or the application is designed in a way which allows treating cheating faults transparently.

5.2 Executing EAs in DGVCS's

There is a growing interest of the community gathered around the parallel and distributed EAs for utilisation of vast resources offered by DGVCS's [21, 24, 41, 88, 129, 144] for time-consuming applications. However, not all available models or optimised problems are suitable for execution in the platform. This section provides general guidelines that should be taken into consideration when choosing DGVCS's.

5.2.1 Is the optimised problem suitable for execution in DGVCS's?

The main criteria for the optimised problem to be considered suitable for execution in DGVCS's are the CPU time required for the evaluation of a single individual and its demands for memory and storage. As mentioned is Section 3.1.3, DGVCS's consist of various computing nodes, typically geographically dispersed, with different network bandwidths and latencies (not to mention the overall performance). Therefore, the evaluation time should be much longer than the time required to send an individual for its fitness value computation. Otherwise, there is no benefit from using the platform, only extending the execution time and increasing the complexity of the software. An ideal example was reported by Desell et al. in [42] from a real-world project (MilkyWay@home) running parallel EAs. There, the evaluation on a high-end CPU takes around an hour, on a slow CPU — days and on a high-end double precision GPU — under 2 minutes.

5.2.2 Which EA execution models are suitable for the platform?

From the models presented in Section 4.3 (and summarised in Table 4.1), two approaches can be dismissed outright from the execution in DGVCS's: the distributed, structured Cellular model and the distributed Island model. The remaining approaches do not pose any problems induced by the platform and related to performance.

Any solutions relying on the rigid relations between the computing resources are at best problematic to implement in a volatile platform provided by DGVCS's. In the case of CEAs (see Sections 4.2.3.3 and 4.3.3), the grid-based structure is hard to maintain. A lot of the computing time would be devoted exclusively to the reconstruction and reconfiguration of the connections between the cells. Therefore, the distributed version of the approach is not suitable for the execution in DGVCS's and the parallel version is preferred.

The distributed Island model has a very limited scalability. As described in Section 4.2.3.2: using too many islands is wasting computational resources and does not give any additional benefits to the evolutionary process [26]. Therefore, implementations adopting the scheme are not able to utilise the potential of DGVCS's. What is more, any plausible crash failures may lead to huge losses of individuals, thus implementing a checkpointing mechanism (see Section 3.3.3) is necessary (which increases not only the complexity of the software but also the execution time). For these reasons, the decentralised version of the approach is preferred for DGVCS's. The islands should be located on the controlled, secure and stable resources with single tasks (like evaluation of the individuals) delegated to the volunteered machines.

5.3 Fault-tolerant aspects of EAs in DGVCS's: related works and open problems

A set of recent studies [37,64,75,86,109,111] illustrate what seems to be a natural resilience of EAs against a model of destructive faults (crash faults, see Section 3.3.1). With a properly designed execution, the system experiences a **graceful degradation** [37,64,86]. This means, that up to some threshold and despite the failures, the results are still delivered. However, it either requires more time for the execution or the returned values are further from the optimum being searched.

There are two types of computing resources in DGVCS's: managed by the platform owner and volunteered. In case of the first type, it is generally assumed that they are always in a "safe state" as proper technical solutions may be adopted to provide 24/7 availability [7]. The volunteered nodes are the main concern. The approach to tolerate failures depends on the specific execution model. However, there are three typical solutions applied in DGVCS's [65] against crash and cheating failures:

1. **Checkpointing** — the state of the computation is periodically stored and if a failure occurs, the execution is restarted from the last, correct save point. Depending on the location of the storage, there are two approaches:
   a) Local — a task is restarted on the same node from the local checkpoint after recovery from the failure.
   b) Centralised — checkpoints are stored on a checkpointing server. After a failure, a task is restarted from the last checkpoint on a new node.
2. **Re-execution** (i.e. **retry**) — a task is executed again on a new resource after a failure.
3. **Replication** (i.e. **redundancy**) — same task is sent to two or more nodes. Either the

first result from a failure-free execution is accepted and stored, or the results from all replicas are collected and the final output is dictated by the majority. This technique has to be combined with at least one from above to deal with a failure of all nodes executing the copies of the task.

Checkpointing may be used in parallel and distributed executions. It introduces some overhead to the computation linked to the process of preparing and storing the state of the application. The method is reasonably effective, however rarely used in the context of EAs as more efficient, problem specific solutions are available — see Sections 5.3.1.1 and 5.3.2.

Replication and re-execution are typically used in tandem for parallel applications. The first one ensures correctness of the results, while the second — that all tasks will be executed. Their application in the context of EAs is described in Sections 5.3.1.2 and 5.3.1.3.

5.3.1 Crash- and cheating-tolerant executions of parallel EAs

As described in Section 4.3.1, two types of nodes are employed in parallel EAs: the master and workers. While executing the algorithm in DGVCS's, the master node resides in the safe environment of the platform owner and the single tasks (like evaluations of individuals) are delegated to the volunteered resources (workers).

First the dynamic populations approach will be presented. It is an ABFT feature of EAs, allowing the population size to change during the execution without significant losses in the quality of the final solution. In particular, this property can be used to tolerate crash failures. This is followed by a description of two, fully fault-tolerant execution schemes. The first one is based on generic techniques available in DGVCS's (and particularly in BOINC), the second one — tailored especially for asynchronous, parallel EAs (empirically validated and used in practice in MilkyWay@home [40, 42]).

5.3.1.1 Dynamic populations

The **dynamic populations** [54, 93, 101, 160] were first introduced in the context of minimising fitness stagnation (countering the lack of progress during the search). In these solutions, the population size is reduced and/or increased in a controlled manner during the algorithm execution. For instance, Fernandez et al. [54] proposed a new operator called **plague**, linearly reducing the number of individuals over time.

However, the approach differs in the context of crash faults experienced in DGVCS's. The process of changing the population size is no longer controlled, but occurs randomly with node or communication failures. At first, it was empirically demonstrated that no special techniques are needed to achieve crash-tolerance in EAs [37, 64]. If a node experienced a failure, the individual being evaluated is simply considered lost and the search continues without it. The main assumption in these works was that once a computing resource becomes unavailable, it will never become available again. This leads to a decreasing population size over time, following the node failures. If the number of individuals was set to compensate the losses at the beginning of the search, then the algorithm experiences a **graceful degradation** — the final result of the computation is equal or close to the fault-free execution.

In [65] the work was extended. In this context, the unavailable resources may become available again. When nodes rejoin the search after recovering form the failures, new individuals are created and added to the population. Ultimately, further improving the graceful degradation of the algorithm. In [65] the new individuals were created randomly. Such

approach improves the diversity of the population, allowing in some cases to find better solutions than the fault-free executions [65]. In [86] additional options were explored, where the new individuals were created following the reproduction scheme or through applying a local search on the existing candidate solutions.

5.3.1.2 Generic solutions from BOINC

The BOINC framework [7] provides a set of generic solutions supporting parallel executions of applications. Its computing model provides **validation** and **retry** mechanisms [10]. The first one is employed to ensure the correctness of the results (returned from the worker nodes), the second one — ensures that each task is eventually executed. Tasks are performed independently on two or more (different) nodes (**redundant** execution, see Section 3.3.3), their outputs are compared, looking for a **quorum**. The quorum is reached if the majority of results returned for a single task are equivalent. In this case, the **canonical** result is marked as valid and nodes participating in its computation are rewarded. Otherwise, more instances of a task are generated as needed until the quorum is reached.

Based on above, both (synchronous and asynchronous) parallel execution schemas described in Section 4.3.1 may be used without any modifications, resulting in a fault-tolerant run of the algorithm. However, if the steady-state reproduction (the asynchronous model) is used, as reported in [42] for the MilkyWay@home: a large amount of computation is wasted on validation. Only a fraction (less than 4%) of evaluated individuals make into the population and as the search progresses it is more difficult to find better solutions (further decreasing the ratio to less than 2%). Therefore, they proposed an improved execution schema for parallel EAs (presented in the subsequent section).

BOINC also contains a mechanism for **adaptive replication** [8]. It is based on a measure of trust: a host gains trust if its results validate, it looses it — when the returned outputs are invalid. This leads to occasional validation of results from trusted hosts. However, the mechanism is unsuitable for executions of EAs, as a single, invalid individual (e.g. with a wrong fitness value) remaining in the population (e.g. when the elitist selection mechanisms are used) may invalidate or decrease the effectiveness of the search [42].

5.3.1.3 Improved BOINC-based approach

As mentioned in the previous section, when the asynchronous parallel model for EA executions is used, only a small fraction (less than 4%) of evaluated results make into the population [42]. This ratio decreases with the progress of the search to less than 2% [42]. On top of that, only 0.5% of the results were reported as invalid in the real-life executions from the MilkyWay@home project [42]. Therefore, a solution limiting the number of required validations (i.e. additional fitness values computations) is desirable.

Desell et al. proposed in [40, 42] to utilise two populations in the search: one containing only validated individuals, the other one — non-validated (but evaluated) ones. A solution after its first evaluation is inserted into the non-validated population (replacing the worst candidate) only if it can improve the validated set. An individual is moved from the non-validated population to the validated one (replacing the worst solution) when it reached a quorum. The approach was tested with two validation strategies: **pessimistic** (Figure 5.3a) and **optimistic** (Figure 5.3b). When the first strategy is used, new individuals are created only from the validated population (using the specified reproduction mechanism) and

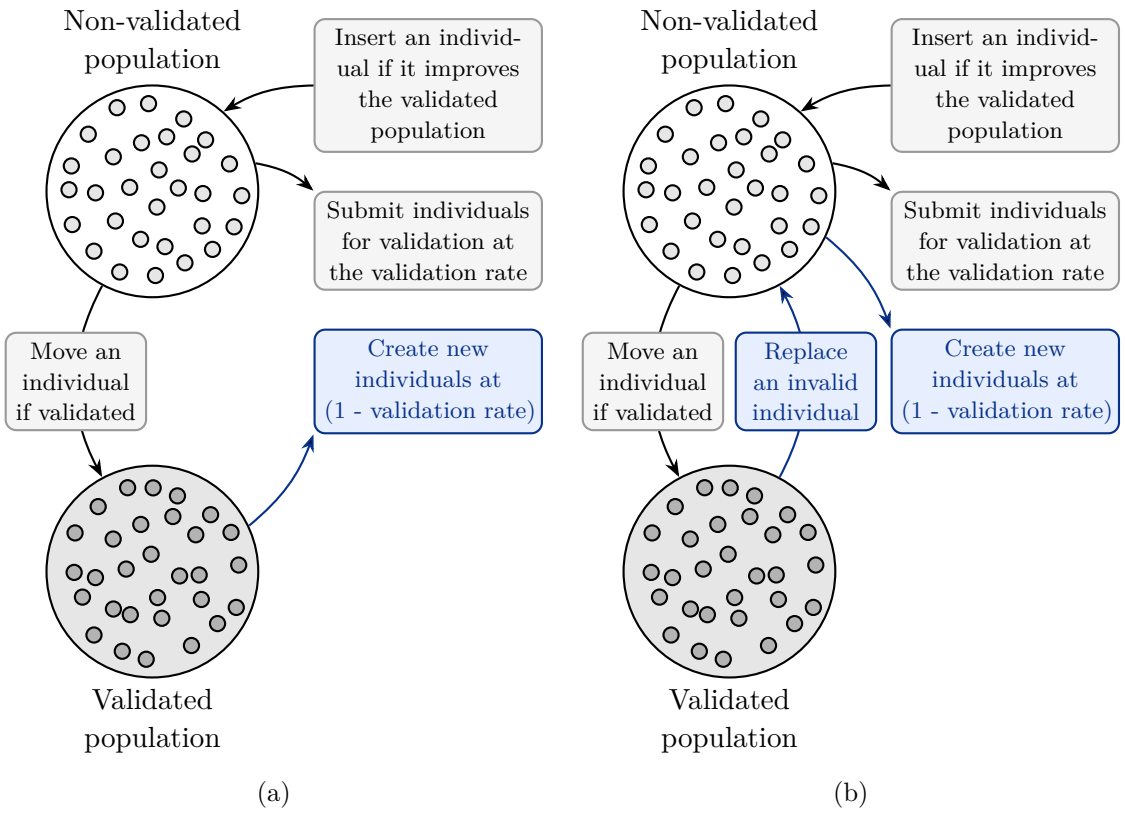


Figure 5.3: An improved BOINC-based approach for fault-tolerant execution of EAs. In (a) the pessimistic strategy, (b) the optimistic strategy.

solutions from the non-validated population are submitted for validation at some (adaptive) **validation rate**. The optimistic strategy is used with an assumption that most of the non-validated results are correct. In this case, new individuals are created using the non-validated population and the solutions from this set are also submitted for validation at some (adaptive) validation rate. If a candidate from the non-validated group was found to be incorrect, it is replaced with a one from the validated population. Additionally, if the newly created individual is never returned evaluated — it is simply considered as lost (adopting the dynamic populations scheme, see Section 5.3.1.1).

If the validation rate is too low, then while using the pessimistic strategy, it may take a lot of time before a good solution is used to create new individuals [40]. On the other hand, in the case of the optimistic approach — the incorrect individuals last longer, generating more (potentially) poor candidate solutions [40]. Another issue is that simple rejecting results without their validation, just because they do not improve the current validated population, attracts cheaters [40]. Constantly reporting bad results gives free rewards for the "done" computation.

To solve these issues, a combination of BOINC's quorum and adaptive replication scheme (see [8]) was adopted for the execution of EAs [40]:

- If there are free worker machines, new individuals (initially with a quorum of 1) are created from the validated population — in the pessimistic approach, or from the non-validated one — if the optimistic strategy is used.

- When the evaluated individual is returned for the first time:
  - If it cannot improve the validated population, it is submitted for validation (its required quorum value is increased to the specified one) with a chance equal to the host's error rate (initially 0.1, multiplied by 0.95 for each correct result, increased by 0.1 — for the incorrect ones). This measure of trust is bounded to the range from 0.1 to 1.
  - If it can improve the validated population, then an attempt to add it to the non-validated population is made and the individual is submitted for validation (increasing its required quorum value).

The optimistic validation leads to faster convergence and shorter execution times [40, 42]. The justification is simple: it is beneficial for the search to quickly use newly found good individuals. This was visible for the results obtained by the pessimistic strategy, which required much higher number of validations [42].

5.3.2 Crash-tolerant execution of distributed EAs

As described in Section 4.3, in distributed environment, there is no distinction between the importance of nodes involved in the computation. In the context of DGVCS's, the resources provided by the platform owner are typically designated only to start and gather the results from the execution. However, an option remains to delegate some nodes to participate in the run of the application at the level equal with the volunteered machines.

As emphasised in Section 5.2.2, only EAs based on unstructured, cellular populations provide the best fit for distributed executions in DGVCS's. Solutions typically relay on P2P overlay networks (see Section 3.2.2) to define relations between the cells. All research in the area is focused on the resilience against node crashes and the performance of the models.

Crash-tolerance is achieved thanks to the decentralised structure of the algorithm [86, 145,179], by the natural extension of the dynamic populations approach (see Section 5.3.1.1) from the parallel environment. Initially, the influence of churn on the evolutionary search was analysed in the area of P2P-based Particle Swarm Optimisation (PSO) using simulations. At first, it was demonstrated through simulated node failures by Scriven et al. in [143] that multi-objective PSO algorithm in failure-prone environment shows resilience to small failure rates. However, the quality of the solution degrades at the high rate of failures. Later on, Banhelyi et al. in [24] showed that churn (simulated through node restarts during the execution) might be helpful in escaping from local optima in the context of P2P PSO and P2P branch and bound algorithm. Finally, the approach was extended by Scriven et al. in [145], exploring different ways to handle the changes in nodes' availability. They demonstrated that the best strategy is to fill gaps with new particles in the approximated pareto front.

The first and currently the only solution designed for massive scalability and crash-tolerance in distributed EAs is the Evolvable Agent Model (EvAg) [85–88].

5.3.2.1 Evolvable Agent Model (EvAg)

Evolvable Agent Model (EvAg) (briefly described in Section 4.3.3) is a complete approach for distributed executions of EAs, first proposed in [87] and further extended in [86, 88]. Its population follows the unstructured, cellular model (see Section 4.2.3.3) with each cell (i.e. Evolvable Agent) acting independently on two layers: **evolution** and **communication**.

On the evolution layer, each agent runs the standard EA loop: parents selection, application of variation operators, evaluation of the offspring, and survivors selection (or in this context, simply — replacement). The execution can be synchronous or asynchronous (see Section 4.3.3), with the latter being a typical setting, additionally increasing the robustness of the solution [143].
The communication layer affects the neighbourhood of each cell, as the population is structured using a P2P overlay network (see Section 3.2). The reference implementation is based on Newscast [82] — a pure P2P gossip (i.e. epidemic) protocol (presented in detail in Chapter 7). In such network, randomness is exploited to virally disseminate information and to maintain connectivity in a self-organised, small-world equilibrium. It emerges from the loosely-coupled and distributed run of the protocol within different and independent nodes (see Appendix C for more details).

Crash-tolerance of the scheme is partially inherited from Newscast which is used to organise the cells. Its epidemic nature provides high resilience and self-healing properties against churn [83, 173]. Depending on the protocol settings, the connectivity between the nodes is maintained even at extremely high node failure rates (reaching 90%) [82].

The participants may join or leave the execution of the algorithm at any moment. Any of the nodes involved in the computation may serve as the entry point for new volunteers. To deal with the joining nodes at the evolution layer, any of the solutions described in Section 5.3.1.1 may be used, namely: new individuals can be created randomly, following the reproduction scheme or though local search performed by the new node. The individuals located at the leaving nodes are simply considered lost.

The solutions described above were analysed by Laredo et al. in [86], based on real-world traces of host availability measured and reported in [91]. Experiments were conducted under two scenarios: nodes that fail, never become available again; and with host-churn (nodes can join and leave the system). The results from the first scenario confirm the findings described earlier in this chapter: low failure rates do not influence the quality of the final solution and the large number of failing nodes damages the optimisation process. However, EvAg loosing up to 70% of its original population, still outperforms the results obtained from a parallel EA with a panmictic population, running in a failure-free environment [86]. In the host-churn scenario, the results again confirm previously presented findings: the quality of solutions is preserved (and in some cases even improved) despite the fluctuations in the number of available resources. "*Furthermore, the hybridization of the P2P approach with the local search policy has provided the most outstanding results: the volunteer system needs to lose up to 90% of the initial resources to diminish the algorithmic performance*" [86].

5.3.3 Open problems

To the best of our knowledge, there existed a single study attempting to provide a theoretical analysis of the impact of cheating-faults on the EA executions. A tentative sketch of proof was initiated in [168], based on the convergence results found in the context of Interacting Particle System (IPS) [39] used to model GAs. Yet the approach suffers from various flaws (see [114] for details) that do not permit concluding rigorously on the convergence of the algorithm.

In Chapter 6, we present the work published in [114], directly inspired by the results found in the seminal article of Rudolph [136], where we addressed the convergence problem in a malicious environment. The conducted study allows identifying critical components of parallel EAs. Additionally, it gives some insights why the genetic material from potentially

erroneous individuals may be used for breeding new solutions (like in Section 5.3.1.3) without preventing the algorithm from converging to the set of optimal solutions.

To the extent of our knowledge, no studies have been conducted on the cheating-tolerance of distributed EAs in DGVCS's. The topic is relatively new and slowly gaining popularity in the community. Therefore, in Chapter 7 we will identify the critical elements of distributed EAs and in particular — EvAg, aiming to extend its resilience to both causes of failures in DGVCS's.

5.4 Summary

In this chapter we presented characteristics of resources availability in DGVCS's and the important term associated with it — CPU availability. The difference between host and CPU availabilities is that the first measurement describes overall uptime of a given resource, where the second — the actual readiness for volunteer computation. In practice, the intervals of the latter may be very short, which was illustrated using the CPU availability trace collected from a real platform — BOINC — executing the SETI@home project. Where this interval was as short as 1 minute for more than 25% of all availability states.

After this, the second major threat in DGVCS's was presented — malicious activities and cheaters. The high anonymity of the platform members allows them to exploit various vulnerabilities build-in within the solution. Such behaviour is additionally motivated by the fact, that Volunteer Computing Systems (VCS's) offer various incentives for their participants. Therefore, malicious users try to obtain these rewards with little or no contribution to the project, which leads to the formalisation of cheating faults and creation of a (Lazy) Cheater Model — mimicking the behaviour.

Having identified main challenges of the platform, we moved to the context of executing EAs in DGVCS's (summarised in Table 5.1). First, we specified general properties of optimisation problems and EA execution models suitable for the platform from the perspective of scalability and performance. Geographical dispersion and diverse hardware of volunteered resources affect efficiency of communication between the machines. Hence, the main factor for suitability of the optimisation problem is the evaluation time of a single solution. It has to outweigh the communication cost and increased complexity of the software.
Regarding the execution models, the parallel model is the safest choice. It offers very good scalability and performance in the platform. It is well analysed empirically, and widely used in practice in volunteer-based projects. Generally, crash failures may be simply ignored up to a certain point. There are also solutions supporting cheating-tolerant executions. The decentralised Island model (the hierarchical parallel EA) may be used as a natural extension of the above, offering better resource utilisation.
Distributed executions of EAs pose the biggest challenge in DGVCS's. Any rigid structure between the volunteered resources is hard to maintain in a volatile environment. Therefore, the structured cellular models are not well suited for the execution in DGVCS's.
The distributed Island model would use a limited number of volunteered nodes, which is dictated by the useful number of islands (too many bring no benefit to the optimisation process, wasting the resources). Additionally, potential crash failures would lead to huge losses of individuals (the whole sub-populations).
The unstructured cellular models have the greatest potential, tolerating the volatile environment very well. However, none of the distributed models were analysed in a malicious environment and their applicability in DGVCS's remains an open problem.

Population structure	**Any**	**Island model**		**Cellular model**	
Execution model	**Parallel**	**Decentralised**	**Distributed**	**Structured**	**Unstructured**
Scalability & performance	✓✓	✓✓✓	XX	✓	✓✓✓
Crash-tolerance	✓✓	✓✓	XX	X	✓✓✓
Cheating-tolerance	✓✓	✓✓	?	?	?
Suitability for DGVCS's	✓✓	✓✓	XX	X	(✓✓)?

Table 5.1: Suitability of EA's execution models for DGVCS's. The scale of assessment: excellent (✓✓✓), very good (✓✓), good (✓), bad (X), very bad (XX), unknown/not yet studied (?).

Chapter 6

Theoretical Foundation of Cheating-Tolerance in Parallel EAs

Contents

The objective of this chapter is to answer the following question mentioned in Section 1.2: *Can we formally analyse in which conditions an EA is expected to converge (or not) towards valid solutions despite the presence of cheating faults?*

To reach the goal, the convergence analysis is performed using a step-wise approach. We first concentrate on the most simple variant of an EA — the (1 + 1) model that is still of theoretical and practical interest as highlighted in Section 4.4.1. In short, in this approach the size of the population is restricted to one individual and the crossover operator is not used. In the second step, the results are extended to regular (population-based) EAs. The ABFT nature of EAs against cheating faults is theoretically demonstrated. The theoretical model of the algorithm introduced in Section 4.4.3 is used throughout the sections of this chapter. Finally, cheating impact on parallel EA computations (typically on top of DGVCS's) is discussed.
Extract of notations in Appendix A might be helpful while following the reasoning presented in this chapter.

6.1 Convergence analysis in a fault-free environment

This section recalls the convergence analysis from [136]. One iteration of the algorithm is divided into two phases: modification and selection. Associated Markovian kernel can be

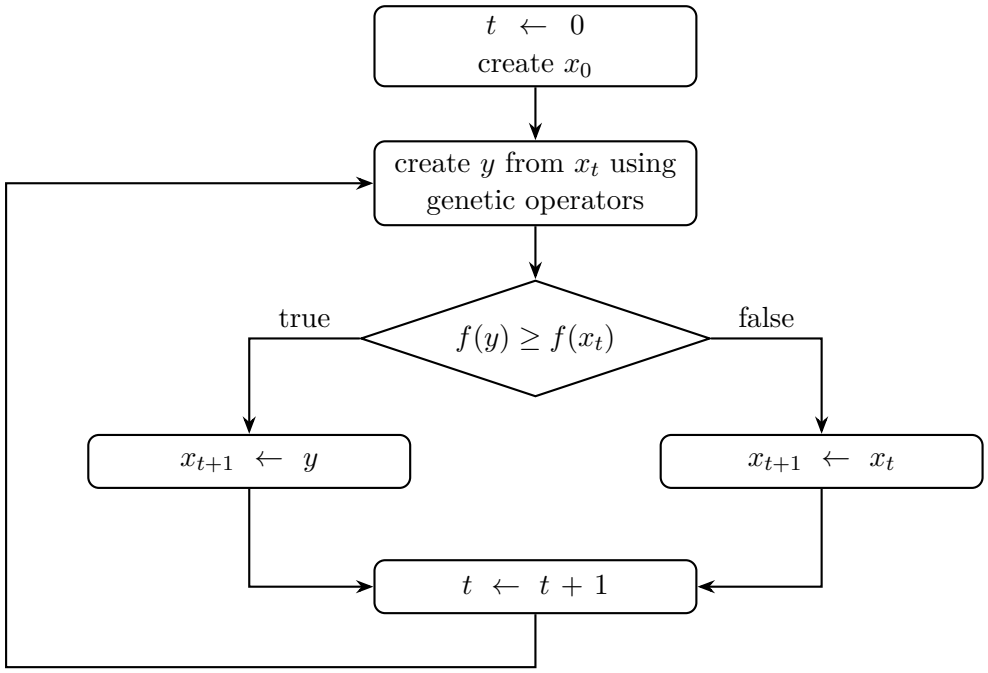


Figure 6.1: Execution scheme of a (1 + 1) EA with the elitist selection.

described as a product kernel defined below [136]:

$$K(x, A) = (K_m K_s)(x, A) = \int_E K_m(x, \mathrm{d}y) \cdot K_s(y, A) \tag{6.1}$$

where K_m is a modification (mutation / mutation-crossover) kernel and K_s is a selection kernel. In all algorithms presented here, the selection phase is performed by an *elitist selection*. It can be implemented in many ways, but all of them have in common one property: the best individual in the current population is not worse than the best from the previous one.

6.1.1 Convergence of (1 + 1) EA

In this case of the (1 + 1) EA, the population consists of a single individual, which in every iteration of the algorithm is used to generate exactly one offspring. Therefore, the state space is $E = \mathcal{M}$. For the illustration of the algorithm execution see Figure 6.1.

It is assumed that the phase of generating the offspring is performed by modification (in this case mutation only) kernel $K_m(x, A)$, which allows transition to any set $A_\epsilon \subseteq \mathcal{A}$ from any state $x \in E$, in short:

$$\forall_{\epsilon>0} \forall_{x \in E} \forall_{A_\epsilon \subseteq \mathcal{A}} K_m(x, A_\epsilon) > 0 \tag{6.2}$$

Additionally, each individual is modified at random independently. The selection phase is described by the elitist selection kernel:

$$K_s(y, A; x) = \mathbb{1}_{A \cap B(x)}(y) + \mathbb{1}_A(x) \cdot \mathbb{1}_{B^c(x)}(y) \tag{6.3}$$

and may be interpreted as follows [136]:

> "*If a state $y \in E$ is better or equal than a state x (i.e. $y \in B(x)$) and also in a set A, then y transitions to the set A, and more precisely to the set $A \cap B(x)$, with probability one. If y is worse than x (i.e. $y \in B^c(x)$) then y is not accepted. Rather, y will transition to the old state x with probability one. But if x was in set the A then y will transition to $x \in A$ with probability one. All other cases have probability zero*".

Combining these two definitions together gives the following product kernel[1]:

$$\begin{aligned} K(x, A) &= \int_E K_m(x, \mathrm{d}y) \cdot K_s(y, A) \\ &= K_m(x, A \cap B(x)) + \mathbb{1}_A(x) \cdot K_m(x, B^c(x)) \end{aligned}$$

Applying restriction to the set A_ϵ of ϵ-optimal solutions, either $A_\epsilon \subset B(x)$, in which case $x \notin A_\epsilon$ ($\mathbb{1}_{A_\epsilon}(x) = 0$), $A_\epsilon \cap B(x) = A_\epsilon$ and

$$K(x, A_\epsilon) = K_m(x, A_\epsilon)$$

or in case of $B(x) \subseteq A_\epsilon$, $x \in A_\epsilon$ (i.e. $\mathbb{1}_{A_\epsilon}(x) = 1$), $A_\epsilon \cap B(x) = B(x)$ and

$$K(x, A_\epsilon) = K_m(x, B(x)) + K_m(x, B^c(x)) = 1 \qquad \text{from (4.7)}$$

Therefore, the Markovian kernel restricted to the set A_ϵ is

$$K(x, A_\epsilon) = \mathbb{1}_{A_\epsilon}(x) + \mathbb{1}_{A_\epsilon^c}(x) \cdot K_m(x, A_\epsilon)$$

which fulfils the preconditions of Theorem 4.4.1 and consequently the algorithm will converge to the global minimum of a real-valued objective function.

6.1.2 Convergence of population-based EA

A population-based EA (considered here) consists of N individuals (from arbitrary set $\mathcal{M}$), which in every iteration of the algorithm are used to generate N offspring (also from the same set $\mathcal{M}$). Therefore, the state space is $E = \mathcal{M}^N$.

Now it is needed to replace previously defined mutation kernel K_m by the corresponding modification kernel, fulfilling the same properties (summarized in Equation 6.2). With a special version of the elitist selection, the selection kernel could be defined as follows:

- If $y \in E$ is in $B(x) \cap A$ then population transitions to A.
- If $y \in E$ is not in $B(x)$ (i.e. the best individual of the population y is worse than the best individual of the population x) then entire population is rejected.

So the population-based selection kernel is identical to the individual-based version defined in Equation (6.3) and a structure of the kernel K defined in case of $(1 + 1)$ EA remains valid.

However, under the usual elitist selection (which is analysed further in this chapter), in second case described above, the best individual is re-inserted — somehow — into the population y yielding $y' = e_{\text{best}}(x, y) \in B(x)$. Where the map $e_{\text{best}} : E \times E \to E$ encapsulates the method to re-insert the best individual of $x \in E$ into y. Consequently, the elitist selection

[1] For the full development of this equation see [136].

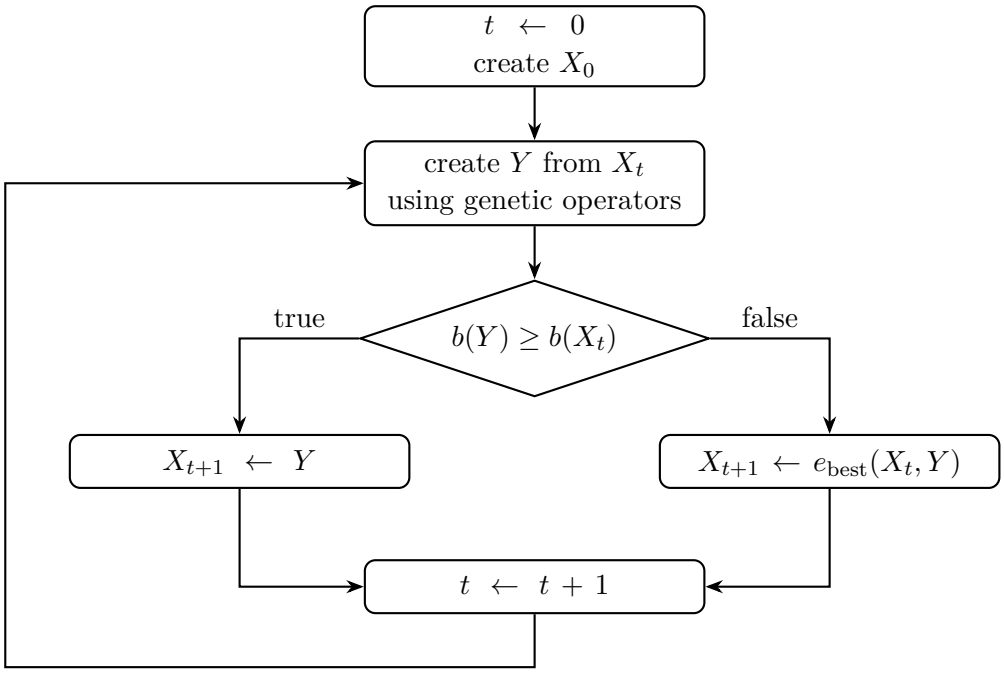


Figure 6.2: Execution scheme of a population-based EA with elitist selection.

kernel becomes:

$$K_s(y, A; x) = \mathbb{1}_{A \cap B(x)}(y) + \mathbb{1}_A(x) \cdot \mathbb{1}_{B^c(x)}(y) \cdot \mathbb{1}_A(e_{\text{best}}(x, y)) \tag{6.4}$$

For an illustration of the algorithm execution, see Figure 6.2.

Combining this elitist selection kernel with the modification kernel (both defined above) gives the following product kernel[2]:

$$\begin{aligned} K(x, A) &= \int_E K_m(x, \mathrm{d}y) \cdot K_s(y, A) \\ &= K_m(x, A \cap B(x)) + \mathbb{1}_A(x) \cdot \int_{B^c(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_A(e_{\text{best}}(x, y)) \end{aligned}$$

Applying restriction to the set A_ϵ of ϵ-optimal solutions, then either $A_\epsilon \subset B(x)$, in which case $x \notin A_\epsilon$ ($\mathbb{1}_{A_\epsilon}(x) = 0$), $A_\epsilon \cap B(x) = A_\epsilon$ and

$$K(x, A_\epsilon) = K_m(x, A_\epsilon) \; .$$

Or in case of $B(x) \subseteq A_\epsilon$, $x \in A_\epsilon$ (i.e. $\mathbb{1}_{A_\epsilon}(x) = 1$), $A_\epsilon \cap B(x) = B(x)$ and since $e_{\text{best}}(x, y) \in B(x) \subseteq A_\epsilon$ (which leads to $\mathbb{1}_{A_\epsilon}(e_{\text{best}}(x, y)) = 1$):

$$K(x, A_\epsilon) = K_m(x, B(x)) + K_m(x, B^c(x)) = 1 \qquad \text{from (4.7)}$$

Therefore, the Markovian kernel restricted to the set A_ϵ is

$$K(x, A_\epsilon) = \mathbb{1}_{A_\epsilon}(x) + \mathbb{1}_{A_\epsilon^c}(x) \cdot K_m(x, A_\epsilon)$$

[2]See footnote 1.

which fulfils preconditions of Theorem 4.4.1 and consequently the algorithm will converge to the global minimum of a real-valued objective function.

6.2 Convergence analysis in a hostile environment

The model of a cheater behavior assumed in our study is the Bernoulli process. At any given point in time a task is either cheated (with a probability $0 \leq \delta_c < 1$) or it is executed without any modifications. The execution history of a task is not taken into account when this decision is made. It is also assumed that there is no cooperation between the cheaters, therefore cheating is performed independently.

6.2.1 Convergence of (1 + 1) EA with cheating at the mutation level

Based on the mutation kernel definition in Equation (6.2), the cheated mutation kernel can be defined as:

$$K_m^{\text{cheat}}(x, A) = \delta_c \cdot \mathbb{1}_A(x) + (1 - \delta_c) \cdot K_m(x, A) \tag{6.5}$$

where δ_c corresponds to the probability of cheating. The above equation means that if cheating occurred (with probability δ_c) and state x was in a set A, then it will stay in this set (lazy cheater model). If cheating did not occur then x will transition to the set A with probability of uncheated version of the mutation. Selection kernel is defined like previously (see Equation (6.3)).

Theorem 6.2.1. *(1 + 1) EA with transition kernels defined above will converge to the global minimum of a real-valued objective function $f : \mathcal{M} \to \mathbb{R}$ with $f > -\infty$ defined on an arbitrary space $\mathcal{M}$, regardless of the initial distribution, despite cheating at the level of mutation.*

Proof.

$$\begin{aligned}
K(x, A) &= \int_E K_m^{\text{cheat}}(x, \mathrm{d}y) \cdot K_s(y, A; x) \\
&= \int_E K_m^{\text{cheat}}(x, \mathrm{d}y) \cdot \mathbb{1}_{A \cap B(x)}(y) && \text{from (6.3)} \\
&\quad + \mathbb{1}_A(x) \cdot \int_E K_m^{\text{cheat}}(x, \mathrm{d}y) \cdot \mathbb{1}_{B^c(x)}(y) \\
&= K_m^{\text{cheat}}(x, A \cap B(x)) + \mathbb{1}_A(x) \cdot K_m^{\text{cheat}}(x, B^c(x)) && \text{from (4.6)} \\
&= \delta_c \cdot \mathbb{1}_{A \cap B(x)}(x) + (1 - \delta_c) \cdot K_m(x, A \cap B(x)) && \text{from (6.5)} \\
&\quad + \mathbb{1}_A(x) \cdot [\delta_c \cdot \mathbb{1}_{B^c(x)}(x) + (1 - \delta_c) \cdot K_m(x, B^c(x))] \\
&= \delta_c \cdot \mathbb{1}_{A \cap B(x)}(x) + (1 - \delta_c) \cdot K_m(x, A \cap B(x)) \\
&\quad + \delta_c \cdot \mathbb{1}_{A \cap B^c(x)}(x) + (1 - \delta_c) \cdot \mathbb{1}_A(x) \cdot K_m(x, B^c(x)) \\
&= \delta_c \cdot \mathbb{1}_A(x) + (1 - \delta_c) \cdot K_m(x, A \cap B(x)) \\
&\quad + (1 - \delta_c) \cdot \mathbb{1}_A(x) \cdot K_m(x, B^c(x))
\end{aligned}$$

Applying restriction to the set A_ϵ of ϵ-optimal solutions, then either $A_\epsilon \subset B(x)$ in which case $x \notin A_\epsilon$ ($\mathbb{1}_{A_\epsilon}(x) = 0$), $A_\epsilon \cap B(x) = A_\epsilon$ and

$$K(x, A_\epsilon) = (1 - \delta_c) \cdot K_m(x, A_\epsilon)$$

Or in case of $B(x) \subseteq A_\epsilon$, $x \in A_\epsilon$ (i.e. $\mathbb{1}_{A_\epsilon}(x) = 1$), $A_\epsilon \cap B(x) = B(x)$ and

$$\begin{aligned} K(x, A_\epsilon) &= \delta_c + (1 - \delta_c) \cdot K_m(x, B(x)) + (1 - \delta_c) \cdot K_m(x, B^c(x)) \\ &= \delta_c + (1 - \delta_c) \cdot [K_m(x, B(x)) + K_m(x, B^c(x))] \\ &= \delta_c + 1 - \delta_c = 1 \qquad \text{from (4.7)} \end{aligned}$$

Therefore, the Markovian kernel restricted to the set A_ϵ is

$$K(x, A_\epsilon) = \mathbb{1}_{A_\epsilon}(x) + \mathbb{1}_{A_\epsilon^c}(x) \cdot (1 - \delta_c) \cdot K_m(x, A_\epsilon)$$

Consequently, (1 + 1) EA with cheating at the level of mutation will converge to the global minimum of a real-valued objective function (according to Theorem 4.4.1). □

6.2.2 Non-convergence of (1 + 1) EA with cheating at the selection level

Mutation kernel is defined like in Section 6.1.1. The elitist selection kernel with cheating becomes:

$$K_s^{\text{cheat}}(y, A; x) = \delta_c \cdot [\mathbb{1}_{A \cap B^c(x)}(y) + \mathbb{1}_A(x) \cdot \mathbb{1}_{B(x)}(y)] + (1 - \delta_c) \cdot K_s(y, A; x) \tag{6.6}$$

where $K_s(y, A; x)$ is defined like previously (see Equation (6.3)). The above equation can be interpreted as follows: if cheating occurred (with probability δ_c) and state $y \in E$ is not worse than state x (i.e. $y \in B^c(x)$) but also in set A, then y transitions to the set A, and more precisely to the set $A \cap B^c(x)$. If y is better than x (i.e. $y \in B(x)$) then y is not accepted and will transition to old state x given that $x \in A$. If cheating did not occur then y will transition to the set A with probability of uncheated version of selection.

Theorem 6.2.2. *(1 + 1) EA with transition kernels defined above may not converge to the global minimum of a real-valued objective function $f : \mathcal{M} \to \mathbb{R}$ with $f > -\infty$ defined on an arbitrary space $\mathcal{M}$, regardless of the initial distribution, due to cheating at the level of selection.*

Proof.

$$\begin{aligned} K(x, A) &= \int_E K_m(x, \mathrm{d}y) \cdot K_s^{\text{cheat}}(y, A; x) \\ &= \delta_c \cdot \Bigg[\int_E K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{A \cap B^c(x)}(y) \qquad \text{from (6.6)} \\ &\quad + \mathbb{1}_A(x) \cdot \int_E K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{B(x)}(y) \Bigg] \\ &\quad + (1 - \delta_c) \cdot \int_E K_m(x, \mathrm{d}y) \cdot K_s(y, A; x) \\ &= \delta_c \cdot [K_m(x, A \cap B^c(x)) + \mathbb{1}_A(x) \cdot K_m(x, B(x))] \qquad \text{from (4.6)} \\ &\quad + (1 - \delta_c) \cdot \Bigg[\int_E K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{A \cap B(x)}(y) \qquad \text{from (6.3)} \end{aligned}$$

$$
\begin{aligned}
&\quad + \mathbb{1}_A(x) \cdot \int_E K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{B^c(x)}(y) \Big] \\
&= \delta_c \cdot [K_m(x, A \cap B^c(x)) + \mathbb{1}_A(x) \cdot K_m(x, B(x))] \\
&\quad + (1 - \delta_c) \cdot [K_m(x, A \cap B(x)) + \mathbb{1}_A(x) \cdot K_m(x, B^c(x))]
\end{aligned}
$$

Applying restriction to the set A_ϵ of ϵ-optimal solutions, then either $A_\epsilon \subset B(x)$ ($\mathbb{1}_{A_\epsilon}(x) = 0$), in which case $x \notin A_\epsilon$, $A_\epsilon \cap B(x) = A_\epsilon$, $A_\epsilon \cap B^c(x) = \emptyset$ and

$$K(x, A_\epsilon) = (1 - \delta_c) \cdot K_m(x, A_\epsilon).$$

Or in the case of $B(x) \subseteq A_\epsilon$, $x \in A_\epsilon$ (i.e. $\mathbb{1}_{A_\epsilon}(x) = 1$), $A_\epsilon \cap B(x) = B(x)$, $A_\epsilon \cap B^c(x) = A_\epsilon \setminus B(x)$ and

$$
\begin{aligned}
K(x, A_\epsilon) &= \delta_c \cdot [K_m(x, A_\epsilon \setminus B(x)) + K_m(x, B(x))] \\
&\quad + (1 - \delta_c) \cdot [K_m(x, B(x)) + K_m(x, B^c(x))] \\
&= \delta_c \cdot [K_m(x, A_\epsilon) - K_m(x, B(x)) + K_m(x, B(x))] + 1 - \delta_c \\
&= \delta_c \cdot K_m(x, A_\epsilon) + 1 - \delta_c
\end{aligned}
$$

Therefore, the Markovian kernel restricted to the set A_ϵ is

$$K(x, A_\epsilon) = \mathbb{1}_{A_\epsilon}(x) \cdot (1 - \delta_c + \delta_c \cdot K_m(x, A_\epsilon)) + \mathbb{1}_{A_\epsilon{}^c}(x) \cdot (1 - \delta_c) \cdot K_m(x, A_\epsilon)$$

There is a probability of losing the optimal solution, such that (1 + 1) EA with cheating at the level of selection may not converge to the global minimum of a real-valued objective function (according to Theorem 4.4.1, the above kernel does not fulfill preconditions for convergence). □

Now the results will be extended to regular, population-based EAs.

6.2.3 Convergence of population-based EA with cheating at the modification level

The Markovian kernels used in this section are defined like previously:

- uncheated modification kernel $K_m(x, A)$ — see Equation (6.2),
- cheated modification kernel $K_m^{\text{cheat}}(x, A)$ — see Equation (6.5),
- elitist selection kernel $K_s(x, A)$ — see Equation (6.3).

Theorem 6.2.3. *Population-based EA with transition kernels defined above will converge to the global minimum of a real-valued objective function $f : \mathcal{M} \to \mathbb{R}$ with $f > -\infty$ defined on an arbitrary space $\mathcal{M}$, regardless of the initial distribution, despite cheating at the level of modification.*

Proof.

$$
\begin{aligned}
K(x, A) &= \int_E K_m^{\text{cheat}}(x, \mathrm{d}y) \cdot K_s(y, A; x) \\
&= \int_E K_m^{\text{cheat}}(x, \mathrm{d}y) \cdot \mathbb{1}_{A \cap B(x)}(y) && \text{from (6.4)}
\end{aligned}
$$

$$
\begin{aligned}
&+ \mathbb{1}_A(x) \cdot \int_E K_m^{\text{cheat}}(x, \mathrm{d}y) \cdot \mathbb{1}_{B^c(x)}(y) \cdot \mathbb{1}_A(e_{\text{best}}(x,y)) \\
&= K_m^{\text{cheat}}(x, A \cap B(x)) && \text{from (4.6)} \\
&\quad + \mathbb{1}_A(x) \cdot \int_{B^c(x)} K_m^{\text{cheat}}(x, \mathrm{d}y) \cdot \mathbb{1}_A(e_{\text{best}}(x,y)) && \text{from (4.6)} \\
&= \delta_c \cdot \mathbb{1}_{A \cap B(x)}(x) + (1-\delta_c) \cdot K_m(x, A \cap B(x)) && \text{from (6.5)} \\
&\quad + \mathbb{1}_A(x) \cdot \int_{B^c(x)} [\delta_c \cdot \mathbb{1}_{\mathrm{d}y}(x) + (1-\delta_c) \cdot K_m(x, \mathrm{d}y)] \cdot \mathbb{1}_A(e_{\text{best}}(x,y)) \\
&= \delta_c \cdot \mathbb{1}_{A \cap B(x)}(x) + (1-\delta_c) \cdot K_m(x, A \cap B(x)) \\
&\quad + \mathbb{1}_A(x) \cdot \delta_c \cdot \int_{B^c(x)} \mathbb{1}_{\mathrm{d}y}(x) \cdot \mathbb{1}_A(e_{\text{best}}(x,y)) \\
&\quad + \mathbb{1}_A(x) \cdot (1-\delta_c) \cdot \int_{B^c(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_A(e_{\text{best}}(x,y))
\end{aligned}
$$

Applying restriction to the set A_ϵ of ϵ-optimal solutions, then either $A_\epsilon \subset B(x)$, in which case $x \notin A_\epsilon$ ($\mathbb{1}_{A_\epsilon}(x) = 0$), $A_\epsilon \cap B(x) = A_\epsilon$ and

$$K(x, A_\epsilon) = (1-\delta_c) \cdot K_m(x, A_\epsilon) \; .$$

Or in the case of $B(x) \subseteq A_\epsilon$, $x \in A_\epsilon$ (i.e. $\mathbb{1}_{A_\epsilon}(x) = 1$), $A_\epsilon \cap B(x) = B(x)$ and since $e_{\text{best}}(x,y) \in B(x) \subseteq A_\epsilon$ (leading to $\mathbb{1}_{A_\epsilon}(e_{\text{best}}(x,y)) = 1$):

$$
\begin{aligned}
K(x, A_\epsilon) &= \delta_c \cdot \mathbb{1}_{B(x)}(x) + (1-\delta_c) \cdot K_m(x, B(x)) \\
&\quad + \delta_c \cdot \int_{B^c(x)} \mathbb{1}_{\mathrm{d}y}(x) + (1-\delta_c) \cdot \int_{B^c(x)} K_m(x, \mathrm{d}y) \\
&= \delta_c + (1-\delta_c) \cdot K_m(x, B(x)) + (1-\delta_c) \cdot K_m(x, B^c(x)) && \text{from (4.6)} \\
&= \delta_c + (1-\delta_c) \cdot [K_m(x, B(x)) + K_m(x, B^c(x))] \\
&= \delta_c + 1 - \delta_c = 1 && \text{from (4.7)}
\end{aligned}
$$

Therefore, the Markovian kernel restricted to the set A_ϵ is

$$K(x, A_\epsilon) = \mathbb{1}_{A_\epsilon}(x) + \mathbb{1}_{A_\epsilon{}^c}(x) \cdot (1-\delta_c) \cdot K_m(x, A_\epsilon)$$

Consequently, the population-based EA with cheating at the level of modification will converge to the global minimum of a real-valued objective function (according to Theorem 4.4.1). □

6.2.4 Non-convergence of population-based EA with cheating at the selection level

The modification kernel is defined like previously, see Equation (6.2). The elitist selection kernel with cheating becomes:

$$
\begin{aligned}
K_s^{\text{cheat}}(y, A; x) = \delta_c \cdot &\left[\mathbb{1}_{A \cap B^c(x)}(y) + \mathbb{1}_{B(x)}(y) \cdot \mathbb{1}_A(x) \cdot \mathbb{1}_A(e_{\text{worst}}(x,y))\right] \\
&+ (1-\delta_c) \cdot K_s(y, A; x)
\end{aligned}
\tag{6.7}
$$

where $K_s(y, A; x)$ is defined in Equation (6.4). For this definition additional map $e_{\text{worst}} : E \times E \to E$ is needed, which encapsulates the method to insert the worst individual of $y \in E$ into x.

Equation (6.7) can be interpreted as follows: if cheating occurred (with probability δ_c) and a state $y \in E$ is worse than a state x (i.e. $y \in B^c(x)$) and also in a set A, then y transitions to the set A, and more precisely to the set $A \cap B^c(x)$. If y is better than x (i.e. $y \in B(x)$), then y is not accepted and will transition to the state composed from the worst individuals of both populations (x and y), given that this state is in the set A. If cheating did not occur, then y will transition to set A with probability of uncheated version of selection.

Theorem 6.2.4. *Population-based EA with transition kernels defined above may not converge to the global minimum of a real-valued objective function $f : \mathcal{M} \to \mathbb{R}$ with $f > -\infty$ defined on an arbitrary space $\mathcal{M}$, regardless of the initial distribution, due to cheating at the level of selection.*

Proof.

$$
\begin{aligned}
K(x, A) &= \int_E K_m(x, \mathrm{d}y) \cdot K_s^{\text{cheat}}(y, A; x) \\
&= \delta_c \cdot \Bigg[K_m(x, A \cap B^c(x)) && \text{from (6.7)} \\
&\quad + \mathbb{1}_A(x) \cdot \int_{B(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_A(e_{\text{worst}}(x, y)) \Bigg] \\
&\quad + (1 - \delta_c) \cdot K_s(y, A; x) \\
&= \delta_c \cdot \Bigg[K_m(x, A \cap B^c(x)) \\
&\quad + \mathbb{1}_A(x) \cdot \int_{B(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_A(e_{\text{worst}}(x, y)) \Bigg] \\
&\quad + (1 - \delta_c) \cdot \Bigg[K_m(x, A \cap B(x)) && \text{from (6.4)} \\
&\quad + \mathbb{1}_A(x) \cdot \int_{B^c(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_A(e_{\text{best}}(x, y)) \Bigg]
\end{aligned}
$$

Applying restriction to the set A_ϵ of ϵ-optimal solutions, then either $A_\epsilon \subset B(x)$ in which case $x \notin A_\epsilon$ ($\mathbb{1}_{A_\epsilon}(x) = 0$), $A_\epsilon \cap B(x) = A_\epsilon$, $A_\epsilon \cap B^c(x) = \emptyset$ and since $e_{\text{best}}(x, y) \in B(x) \subseteq A_\epsilon$ (leading to $\mathbb{1}_{A_\epsilon}(e_{\text{best}}(x, y)) = 1$):

$$
K(x, A_\epsilon) = (1 - \delta_c) \cdot K_m(x, A_\epsilon) \, .
$$

Or in the case of $B(x) \subseteq A_\epsilon$, $x \in A_\epsilon$ (i.e. $\mathbb{1}_{A_\epsilon}(x) = 1$), $A_\epsilon \cap B(x) = B(x)$, $A_\epsilon \cap B^c(x) = A_\epsilon \setminus B(x)$ and

$$
\begin{aligned}
K(x, A_\epsilon) &= \delta_c \cdot \Bigg[K_m(x, A_\epsilon \setminus B(x)) \\
&\quad + \int_{B(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{A_\epsilon}(e_{\mathrm{worst}}(x, y)) \Bigg] \\
&\quad + (1 - \delta_c) \cdot \Bigg[K_m(x, B(x)) \\
&\quad + \int_{B^c(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{A_\epsilon}(e_{\mathrm{best}}(x, y)) \Bigg] \\
&= \delta_c \cdot \Bigg[K_m(x, A_\epsilon \setminus B(x)) \\
&\quad + \int_{B(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{A_\epsilon}(e_{\mathrm{worst}}(x, y)) \Bigg] \\
&\quad + (1 - \delta_c) \cdot [K_m(x, B(x)) + K_m(x, B^c(x))] \\
&= \delta_c \cdot \Bigg[K_m(x, A_\epsilon) - K_m(x, B(x)) \\
&\quad + \int_{B(x)} K_m(x, \mathrm{d}y) \cdot \mathbb{1}_{A_\epsilon}(e_{\mathrm{worst}}(x, y)) \Bigg] \\
&\quad + 1 - \delta_c \qquad \text{from (4.7)} \\
&\leq 1 - \delta_c \cdot [1 - K_m(x, A_\epsilon)]
\end{aligned}
$$

The Markovian kernel restricted to the set A_ϵ (both cases combined together) has a probability of losing optimal solution, such that population-based EA with cheating at the level of selection may not converge to the global minimum of a real-valued objective function (according to Theorem 4.4.1, the above kernel does not fulfill preconditions for convergence). □

6.3 Applying the results in parallel executions of EAs

The analysis proposed in this chapter shows that the validity of the selection mechanism is crucial to achieve the convergence of EAs. Typically, the parallel execution of an EA is organised in the master/worker model (see Section 4.3.1) with a delegation of single tasks (in the most cases: the evaluation of individuals).

In this approach, the main EA loop (including the selection) is performed by the master node which is a trusted resource in a DGVCS (therefore, never cheating). The breeding and the evaluation of new individuals can be executed by the workers. Even though these machines (among which cheaters can be located) are not directly responsible for the selection process (the parents or the survivors), they can still influence it. This can be achieved by altering the fitness value assigned to a particular individual. For instance, in

case of minimisation of the objective function $f : \mathcal{M} \to \mathbb{R}$, the fitness value can be reversed $f^{\text{cheat}}(x) = 1/f(x)$. This way, the worse genetic material will be chosen in favour of a better one during the selection process.

In case of the elitist selection considered in the analysis, at least the "best so far" member of the population has to be treated specially. It has to be assured that in the next generation, the new best individual is indeed not worse than the previous one (despite of possible cheating). The aforementioned can be achieved by re-evaluating the candidates for the new "best so far" before updating the population.

A direct application of the results from this chapter can be found in the execution schemes presented in Section 5.3.1.3. The assessment of the search progress (and in particular the survivor selection) is made using only the verified (re-evaluated) individuals. However, as Desell et al. empirically demonstrated in [40, 42], the non-validated results can be used for breeding new candidate solutions (which is equivalent to possible cheating at the modification level), obtaining faster convergence and shorter execution times.

6.4 Summary and perspectives

In this chapter, we proposed a formal analysis that permits to conclude on the robustness (or non-robustness) of EAs when executed in a parallel environment subjected to malicious acts. Whereas some preliminary studies illustrate the remarkable resilience of EAs against crash failures, the enclosed convergence results offer new insights on the subject in the presence of cheaters. More precisely, the proposed analysis permits concluding formally on the robustness (or non-robustness) of parallel EAs. Table 6.1 summarizes the contributions of this thesis from this perspective.

The fact that there exists some cases where an EA *always* converges despite the presence of cheating faults is quite encouraging. This will promote the usage of EAs in the future developments around distributed computing platforms such as Desktop Grids and Volunteer Computing Systems or Cloud systems where the resources cannot be fully trusted. In particular, our work shows that the modification step of an EA can be "safely" executed on the untrusted workers without any special protection: if cheating is present at this level, it will not affect the convergence of the optimisation search towards valid and (hopefully) optimal solutions. In this sense, EAs can be considered to have an Algorithm-Based Fault Tolerance (ABFT), hence to be fault-tolerant without any extensions. Alternatively, our study also highlights that as soon as cheating happens at the selection level and no special validation mechanisms are used, there is a chance that the algorithm will *not* converge. This means that in this case, additional measures have to be introduced.

Finally, the convergence towards valid and optimal solutions despite the presence of cheating faults does not mean that the optimisation search will end within a reasonable execution time (compared to a fault-free environment). Nevertheless, the existing empirical studies indicate that for small cheating rates (less than 0.5%) it is beneficial to use the non-validated genetic material in the process of creating new candidate solutions. In future works, it would be interesting to analyse theoretically the impact of cheating on the execution time which combined with this study would create a clear guideline for the practical applications.

	Robustness Questions [172]
Q1	What behavior of the system makes it robust?
Q2	What uncertainties is the system robust against?
Q3	Quantitatively, exactly how robust is the system?

	Algorithm	**(1+1) EA**		**Population-based EA**	
Q1	**FT Mechanism**	None		None	
	Fault Model	**Cheating fault**		**Cheating fault**	
Q2	**Attack Model**	Bernouilli lazy cheater with cheating probability δ_c			
	EA step affected	Mutation	Selection	Modification	Selection
Q3	**EA Convergence**	✓	✗	✓	✗
	EA Robustness	Robust (ABFT) ($\forall\delta_c$, see Theorem 6.2.1)	Not-robust ($\forall\delta_c$, see Theorem 6.2.2)	Robust (ABFT) ($\forall\delta_c$, see Theorem 6.2.3)	Not-robust ($\forall\delta_c$, see Theorem 6.2.4)

Table 6.1: Summary of the robustness (or non-robustness) of EAs in the presence of cheating faults.

Chapter 7

Towards Cheating-Tolerance in Distributed EAs

Contents

In the previous chapter, an in-depth formal analysis of the cheating-tolerance of the parallel EAs has been presented. It included proofs of convergence (or not) towards the optimal solutions when the execution is performed on top of a DGVCS subjected to malicious acts. Here we would like to extend our investigation in the direction of distributed EAs. More

precisely, we propose to analyse Evolvable Agent Model (EvAg). The solution relies on a gossip P2P protocol named Newscast, used to define neighbourhoods in the evolution and the communication layers. As introduced in the Chapter 5, it is the only distributed EA model able to utilise vast computing resources offered by DGVCS's. Moreover, the approach has been already analysed and proved to be effective in the presence of churn (the volatility of resources).

This chapter starts with an introduction of the necessary conditions for convergence of EvAg in a hostile environment. Independently on the approach used at the evolution layer, cheating-tolerance of the communication protocol is necessary to guarantee the successful execution of the search. Due to the lack of suitable, cheating-tolerant replacement for Newscast, the rest of the chapter is devoted to locate the problem and to find a way to solve it. First, the malicious acts and their impact at the communication layer are formalised with an introduction of the simulation environment. Then, the vulnerabilities in Newscast are located through a data flow analysis. This is followed by the development of an effective cheating scheme able to exploit the flaws in the original design. Finally, the solution towards countering the attack is presented.

7.1 Cheating-tolerance of Evolvable Agent Model (EvAg)

As introduced in Section 5.3.2.1, Evolvable Agent Model (EvAg) works on two layers: evolution and communication. Each cell (i.e. Evolvable Agent) acts independently on both layers, where the first layer heavily depends on the latter. It is caused by the definition of the neighbourhood of each cell, which is determined by the network structure (i.e. links between the nodes) obtained from the execution of Newscast [82] — a pure P2P gossip protocol. Cheating-tolerance of the whole scheme may be achieved only if this property is provided on both layers. Therefore, the analysis will be made in both directions.

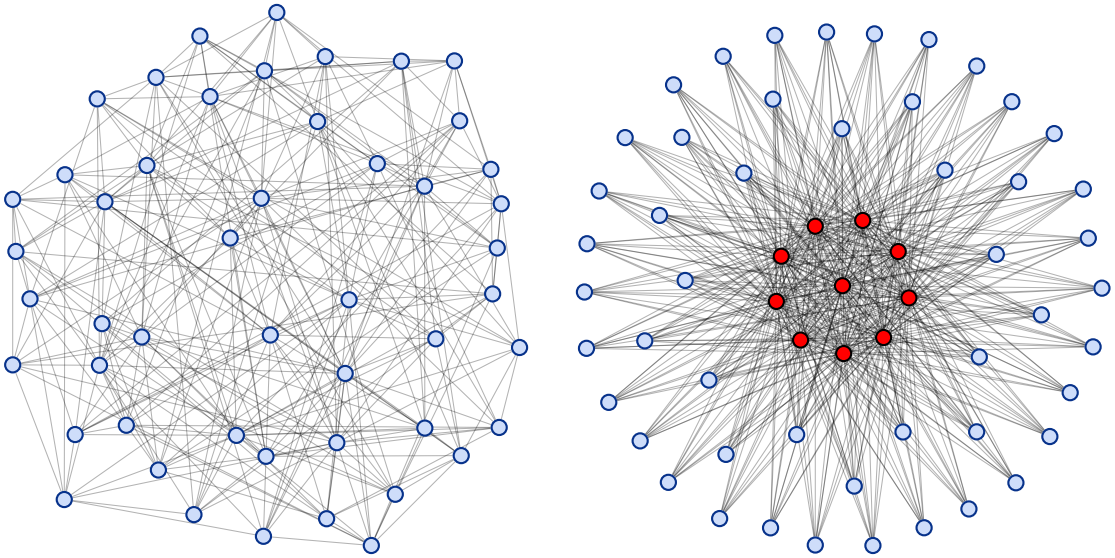
7.1.1 Cheating-tolerance at the evolution layer

As detailed before (in Section 5.3.2.1), each cell (in EvAg) runs the standard EA loop at the evolution layer:

1. Parents are selected from the neighbourhood.
2. An offspring is created through the application of the variation operators on the parent solutions.
3. The newly created solution is evaluated.
4. The individual stored in the cell is possibly replaced by the offspring.

Regardless of whether the replacement is synchronous or asynchronous, in a standard distributed execution, non of the above operations are delegated to other nodes. Consequently, on the current layer, the only "attack vector" for a malicious user is to provide an invalid (i.e cheated, corrupted) individual with an increased chance of being chosen as a parent. For instance, it could be the worst solution encountered by the cheater (during the current execution) with the reversed fitness, or simply: a random specimen with a high objective value.

Therefore, a decision has to be made: if parents should be validated or not (i.e. evaluated again, locally at the current cell). This yields two strategies for validation: pessimistic and optimistic, similarly to the improved BOINC-based execution scheme for parallel EAs (Section 5.3.1.3). However, adapted to the level of a single, independent cell.



(a) (b)

Figure 7.1: The worst case scenario with a cheating-intolerant communication layer for EvAg. In (a) a snapshot of the population structure created by the fault-free execution of a P2P protocol, (b) a snapshot of the population structure altered by 10 malicious users. Cheaters control fully all neighbourhoods at the latter stage.

7.1.1.1 Pessimistic strategy: validate the parents

Validation of all individuals obtained from the neighbourhood of a cell would waste a lot of CPU time. For this reason, only the selected parents (or the parent, if the individual currently stored in the cell is chosen or selected in advance) should be re-evaluated in the pessimistic strategy. If an invalid solution is detected, it is removed from the neighbourhood set and the selection with validation is performed again. When all individuals obtained from the neighbourhood proved to be invalid, it indicates a serious problem: the node is faulty, the execution is attacked by malicious users or many nodes are failing. Hence, the situation should be reported to the owner of the node/nodes and/or the owner of the platform.

Assuming the fault-free state of the node, the above scheme allows progression of the search at the level of a single cell despite cheated individuals present in its neighbourhood. However, the global success of the optimisation depends on the communication layer. Despite the fault-tolerant execution of the evolution at each cell, the overall progress of the search may be crippled or even stopped. For instance, a group of malicious users may try to takeover the network, concentrating all the connections on them (forming a communication hub, see Figure 7.1). To stop the progress of the search, they do not have to send corrupted individuals — it is enough to provide all the time the same genetic material. This would cause the stagnation, and possibly stop the optimisation process at a local optimum. Additionally, in the worst case, the cheaters may leave at the same moment, splitting completely the population into single cells.

7.1.1.2 Optimistic strategy: no validation (an ABFT approach)

In the optimistic strategy, no validation is performed. The creation of the offspring, its evaluation and the replacement mechanism are executed locally. Therefore, the individual stored in the cell and the newly created one are bound to be valid at all times within the boundaries of the node (assuming that it is in a fault-free state). It should also be pointed out that even poor solutions have a potential to create good individuals.

However, if the search will progress globally (at the population level) or at the level of the cell, depends on many factors. Most of them are related again to the protocol being used at the communication layer and the hardware involved in the computation. In a particular case, it would be possible to estimate (or at least measure empirically) the expected number of cheated individuals in the neighbourhood and their persistence. This would allow assessing, if the approach is viable or not. Yet, as will be demonstrated later in this chapter, the protocol used in the original solution is not cheating-tolerant and there is no suitable replacement for it. Hence, we decided to dedicate our efforts to analyse the communication scheme in order to locate the cause of the problem and propose some methods to remove the issues, leaving the optimistic strategy as an open question for future research.

7.1.2 Cheating-tolerance at the communication layer

As previously identified, cheating-tolerance at the communication layer is crucial for a successful execution of EvAg regardless of the validation strategy of individuals. The cheating-tolerant evolution layer alone, does not guarantee reaching the optimal solution (even with properly defined building components of the EA, see Section 4.2). Therefore, now we will present in details the original solution, analysing its resilience to the potential, malicious activities.

7.1.2.1 Newscast: details and cheating-intolerance of the scheme

Newscast is a gossip protocol proposed by Jelasity and van Steen in [82] for interconnecting large-scale distributed systems. Without any central services or servers, the protocol differs from other similar approaches by its simplicity. Membership management follows an extremely simple protocol. In order to join the system, a node only needs to contact a connected node from which it gets a list of neighbours. Whereas to leave — it requires to stop communicating for a predefined time. The dynamics of the system follow a probabilistic scheme able to keep a self-organized equilibrium. Such an equilibrium emerges from the loosely-coupled and decentralized run of the protocol within the different and independent nodes. The emerging graph behaves as a small-world topology allowing a scalable information dissemination (see Section 2.4). Despite the simplicity of the scheme, Newscast is particularly fault-tolerant against crash failures and exhibits a graceful degradation without requiring any extra mechanism other than its own emergent behaviour [173].

Algorithms 7.4, 7.1 and 7.2 show the pseudo-code of the protocol. Each node keeps its own set of neighbours in a cache (i.e. a view) containing $c \in \mathbb{N}$ entries, referring to c other nodes in the network without duplicates. Each entry provides a reference to a node, a time-stamp of the entry creation (allowing replacement of old items) and optionally application-specific data (see Figure 7.2).

There are two different tasks that the algorithm carries out within each node. The active thread (Algorithm 7.1) which pro-actively initiates a cache exchange once every cycle (one

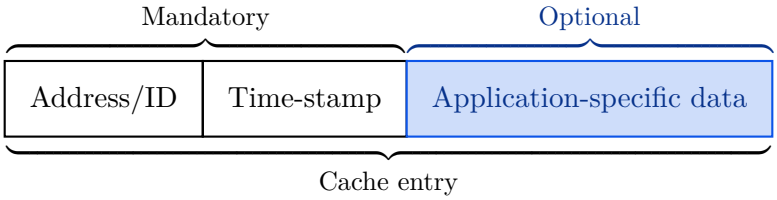


Figure 7.2: Cache entry in Newscast. The address and time-stamp are mandatory. The first field identifies the creator of the entry, the second — the time of creation used in the mechanism responsible for replacement of old items. Application-specific data can be optionally embedded.

Algorithm 7.1: Active thread of Newscast.

while true **do**
- wait t_r;
- $node_j \leftarrow$ select a node from $cache_i$;
- send $cache_i$ and $data_i$ to $node_j$;
- receive $cache_j$ and $data_j$ from $node_j$;
- $cache_j \leftarrow$ `Update`($node_i$, $node_j$, $cache_j$, $data_j$);
- $cache_i \leftarrow$ `Aggregate`($cache_i$, $cache_j$);

Algorithm 7.2: Passive thread of Newscast.

while true **do**
- wait until $cache_k$ and $data_k$ is received from $node_k$;
- send $cache_i$ and $data_i$ to $node_k$;
- $cache_k \leftarrow$ `Update`($node_i$, $node_k$, $cache_k$, $data_k$);
- $cache_i \leftarrow$ `Aggregate`($cache_i$, $cache_k$);

cycle takes t_r time units) and the passive thread (Algorithm 7.2) that waits for data-exchange requests.

Every cycle, each $node_i$ initiates a cache exchange. It selects randomly a neighbour $node_j$ from its $cache_i$ with uniform probability. Then, the nodes $node_i$ and $node_j$ exchange their caches and data (respectively $data_i$ and $data_j$). At $node_i$, the entry from $cache_j$ referring to $node_i$ is substituted with a new item containing the address of $node_j$ and its data $data_j$ (Algorithm 7.3). Conversely, the same operation is performed at $node_j$. Eventually, the caches are merged at both nodes following the aggregation function (Algorithm 7.4). It consists of picking the freshest c items from both caches to form a single cache. Since this function is applied at both nodes (the one initiating the request and the one serving the request), the result is that $node_i$ and $node_j$ will have some entries in common in their respective caches.

Newscast was not designed to be cheating-tolerant and as our preliminary experiments showed — it is not. The problem is located at the aggregation function (i.e. the caches merging operation). Even a single node, exhibiting byzantine behaviour — deliberate or not — flooding the network with incorrect entries (filled with non-existing addresses/IDs) is able to disconnect some fault-free machines. The issue is further amplified by the protocol mechanics, as the corrupted information spreads in the network, leading to collateral damage in other parts of the communication graph.

Algorithm 7.3: Function for updating the cache content.

```
// a cache is a set of c entries
function Update(node_a, node_b, cache_b, data_b) is
    newEntry ← create a new entry using the address of node_b and data_b;
    cache_result ← cache_b \ {entry reffering to node_a} ∪ {newEntry};
    return cache_result;
```

Algorithm 7.4: Cache aggregation/merge function.

```
// a cache is a set of c entries
function Aggregate(cache_a, cache_b) is
    cache_result ← cache_a ∪ cache_b;
    keep the c freshest items in cache_result according with the time-stamp;
    return cache_result;
```

7.1.2.2 No suitable, cheating-tolerant replacement for Newscast

The problem of malicious users is not new in the context of P2P networks and in particular — gossip protocols. There are many existing attempts to solve it. Cheating-tolerance (or more general — byzantine fault tolerance) is usually achieved at the cost of limited gossip performance or loss of other (desirable) properties, like decentralisation or scalability.

Generally, solutions are based on **cryptography** or **operational constraints**. In the first case either the centralisation is re-introduced into the scheme (forming an unstructured-centralised or hybrid-P2P overlay network, see Section 3.2.2) or the local view of each peer becomes only a simulated feature, as members of the network have to possess and maintain the full, global view (i.e. information about all active nodes). When it comes to introducing the operational constants, the available solutions are simply focused on limiting the number of gossips (send and received). We now proceed to a short overview of the concrete solutions found in the literature of the subject.

Minsky and Schneider in [107] proposed **path verification protocols**, based on the assumptions that it is possible to determine a sender of each message and that no data alteration is possible during its transit over the network. The solution allows tolerating fixed amount (t) of faults.
First, the authors introduced a **direct verification** scheme, where a message is accepted when it is received directly from the sender or when $t + 1$ of the neighbours accepted it already. As a consequence, the initial spread of a fresh update is relatively slow. However, the speed of information dissemination increases over time, as more and more nodes accept the gossip, i.e. the probability of contacting someone who already accepted the message grows over time.
The **path verification** protocols extend the previous idea. Each message contains additionally the path which it travelled through the network. In this case, a message is accepted only if it arrived directly or from $t + 1$ disjoint paths. This allows not yet verified gossips to spread much further within the network, in turn increasing the rate at which the messages are accepted. However, storing and managing all encountered updates combined with their paths require substantial storage and computational resources.

JetStream [131], Fireflies [89] and Veracity [148] are based on principals found in social networks. The first two utilise mechanisms for predictable, pseudo-random, and verifiable

gossiping. Such approach combined with ignoring unexpected connections, limits the amount of gossips between the nodes. Additionally, each peer in Fireflies stores the global view on the network and uses only a small part of it for the protocol execution. Despite the embedded cryptographic solutions, scalability and reliability of gossip is maintained up to a certain point. Frequent malicious activities increase the computational effort associated with communication.
In Veracity on the other hand, a slightly different approach is used. The protocol is executed on top of a fully functional DHT service, increasing the complexity of the solution. Messages are accepted, when they are checked by a (small) set of other nodes from the DHT network. However, the base of the protocol opens additional directions of attacks [84].

Puppetcast [16] is a protocol based on a re-introduced centralisation. Essentially, each member of the network holds **two partial views**:

1. A signed, constant view — received from the central server and used to respond to incoming connections. It remains static until the next contact (update or re-registration) with the central authority.
2. A local, modifiable view — used for the actual execution of the protocol, updated with new descriptors at each view exchange.

This protocol is efficient at maintaining connectivity between the members of the network if the central server is operational and reachable at all times. However, it forms a single point of failure and the main target for attacks. If the additional data is signed with the view, then the information spread is greatly limited. If it is not, then it has to be fetched from the network before each use, increasing the amount of traffic. What is more, such solution would not be very efficient at high churn rates (i.e. with highly volatile resources). Its effectiveness in such situation greatly depends on the frequency of exchange of the signed view.

An interesting implementation of gossiping was proposed by Bortnikov et al. in [23]. They performed a theoretical analysis of a bias in the local view of each peer caused by the malicious activities. This lead to creation of Brahms protocol, in which an additional sample of the nodes was introduced (comparing to the standard scheme). This set is built from all descriptors received during the view exchanges, using min-wise independent permutations [23]. Such construct ensures uniform distribution of the sample regardless of the frequency and order in which the addresses are collected. Random elements from this set are added to the view during its update. The approach proved to be effective in a testing environment, however in a real-world setting, when the churn is present, the uniform random sample is impossible to achieve [84]. This is caused by the method of its construction, which heavily depends on a stable environment.

The last protocol described here, is a prestige-based solution proposed by Jesi et al. in [84]. It has a unique feature in comparison to the previously described schemes — an **exploratory view exchange**. Each node, during its synchronisation cycle, additionally to following the standard send-receive scheme, sends a predefined number of **requests for gossip** to randomly chosen peers. If B receives such message from A, it has to respond with its view, for which A sends its own. Lists of descriptors collected this way are used to compute and maintain local statistics on neighbourhoods of each node. This allows identification of structural abnormalities in the neighbourhoods (e.g. hubs are "*over-represented in the partial views*" [84]) and blocking responsible peers from communication. As suggested by authors, like in the previous case (Brahms), it is possible to build a sample of network members, which in case of disconnection or view pollution may be used to restore full connectivity and

performance. However, the solution might be broken by sending the views containing old or random addresses (and in a realistic scenario — most probably belonging to inactive/non-existing nodes). In such case, the efficiency of the communication would be significantly crippled, increasing the overhead associated with maintenance of the statistics and the local repository of known nodes.

None of the above solutions would be able to support the execution of Evolvable Agent Model (EvAg) in a VCS. Only Puppetcast and the prestige-based protocol may partially exhibit the small-world properties of the communication graph, which are the most desired for EvAg. Without such feature, the optimisation performance of the model would be decreased (for example, see Appendix B). Additionally, the Puppetcast does not adopt well to high-churn rates which are present in DGVCS's and the prestige-based protocol may be broken by the (lazy) cheater model (see Section 5.1.2.2) used in this work. Therefore, we decided to extend the original solution, tailoring it for distributed execution of EAs, in particular — EvAg.

7.2 Towards cheating-tolerance of Newscast

Newscast protocol was designed with emphasis on the ease of membership management. It was intended to interconnect vast amount of computing resources in a highly dynamic, decentralized P2P environment. Every client (node) can join and leave the network at any time without prior notification. However, the simplicity of the protocol leaves space for abuse.

The biggest problem is the assumption that the cache content received during the message exchange is correct. Solving this issue is not trivial. Direct verification of the cache entries is infeasible, as it would waste many communication cycles. Existing solutions limit the flexibility and scalability of the approach (as reviewed in the previous section), focused on keeping all the nodes connected together. Nevertheless, this strict requirement could be relaxed for distributed executions of EAs.

Evolutionary Algorithms tolerate loss of genetic material during the execution (see Section 5.3.1). In particular, in EvAg up to 90% of the resources may be lost before the algorithmic performance of the solution is diminished [86]. If the pessimistic validation strategy is used at the evolution layer (see Section 7.1.1), the effort at the communication layer can be shifted towards keeping the largest number of non-malicious nodes connected together for as long time as possible. In such approach, the malicious resources would limit the computing performance of affected nodes (immediate neighbours and the ones to which the cheated information spread). Essentially, both types would act as crashed or inefficient machines.

7.2.1 Formalisation of cheating and experimental setup

Before proceeding to examine cheating faults in the context of Newscast, it is necessary to specify the background of the analysis. In the first place, the kind of cheaters and their activities must be clearly defined. Then, the method to measure the impact of cheating is required. Finally, the approach taken to obtain the data used in the whole analysis has to be outlined.

7.2.1.1 Malicious users and cheating faults

As mentioned earlier, the problem of malicious users in the context of P2P protocols is not new (see Section 7.1.2.2). The cheaters may act individually or in the organised groups, attracted in both cases by various incentives (see Section 5.1.2). Until the end of this chapter, we assume that the malicious users act independently and without cooperation, following the (Lazy) Cheater Model introduced in Section 5.1.2.2.

Messages of the protocol, exchanged between the nodes, are the obvious target of cheaters. We assume that the tampering is performed directly by the cheating nodes and not in transit. This can be achieved either by altering the message just before it is sent or by implementing the malicious client of the protocol. As explained before, the information is assumed to follow the correct (and expected) format to avoid fast detection and removal from the network. Therefore, the goal is to alter the content of the message.

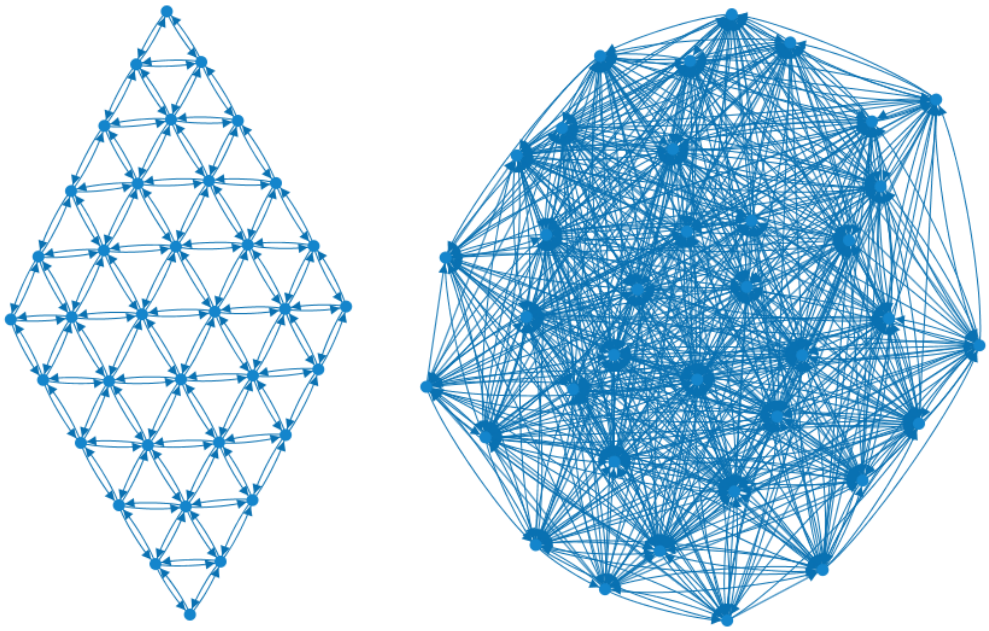
In Newscast, messages contain the cache of each node (its full copy), containing $c \in \mathbb{N}$ entries (see Section 7.1.2.1). Each item holds the time of its creation, the address (i.e. ID) of a node and optionally application-specific data. In the remainder of this chapter we assume that the cache entries contain only the first two informations. Therefore, the goal of a cheater is to alter the address and the creation time of each entry. Obviously to provide the best possible spread of the cheated items in the network, the time-stamps should be set to the most recent time. Sending the message filled in with entries containing only the cheater address is pointless, as uniqueness of the addresses in the cache is easy to verify. Hence, the best approach for a "lazy" cheater is to set all the addresses to random ones. This way, most likely the entries will refer to non-existing peers (for instance, IPv6 uses 128-bit addresses giving approximately 3.4×10^{38} possible values).

7.2.1.2 Impact of cheating faults on the connection graph

The presence of cheated entries in the caches of the peers leads to the decreasing number of valid links between non-malicious nodes. To analyse this effect and its consequences, we need to introduce some preliminary definitions.
Let $G_t = (V_t, A_t)$ be a directed connection graph at a given time t, consisting of the set of vertices V_t and the set of arcs A_t between them. V_t corresponds to the set of nodes in the network at the time t. An arc $a = \langle \overrightarrow{v_i, v_j} \rangle \in A_t$ connecting the vertex (or the node) $v_i \in V_t$ with the node $v_j \in V_t$, reflects the fact that v_j is in the cache (i.e. the view) of v_i. The cache size of each node remains constant within the graph G_t, i.e. $|cache_i| = c \ \forall_{v_i \in V_t \wedge t \geq 0}$. Execution of the protocol leads to a series of graphs G_t, given an initial graph G_0.

For the sake of simplicity, we will assume that the number of non-malicious nodes in the network is constant and equal to $n_{\text{honest}} \ \forall_{t \geq 0}$. In addition, there are $n_{\text{cheating}} \ \forall_{t \geq t_c}$ malicious nodes i.e. cheaters, joining the network at the time t_c (all at the same simulation step). This leads to a partition of the set of vertices in V_t between non-malicious (i.e. honest) nodes and cheaters. Thus, $V_t = V_t^{\text{honest}} \cup V_t^{\text{cheating}}$, where $|V_t^{\text{honest}}| = n_{\text{honest}}$ and $|V_t^{\text{cheating}}| = n_{\text{cheating}}$.

An important concept to measure the robustness of the protocol against cheaters is the size of the connected components of non-malicious nodes in G_t. Let G_t^{honest} be a subgraph of G_t induced by the vertex set V_t^{honest} (see Definition 2.1.7). A j-th (weakly) connected component $\hat{\mathcal{C}}_j$ in G_t^{honest} is a maximal subgraph of G_t^{honest} such that every node in $\hat{\mathcal{C}}_j$ is reachable from every other node in $\hat{\mathcal{C}}_j$ (see Definitions 2.2.3 and 2.2.4). In the sequel, $\hat{\mathcal{C}}_t^{\max}$ will denote the connected component of maximum size (the giant component, see Definition 2.2.6) at a given



(a) (b)

Figure 7.3: An example of bootstrap of the Newscast network. In (a) a sample network consisting of 36 nodes at the initial step of the simulation, in (b) the same network after bootstrap of Newscast (with the cache size set to 20).

time t. Obviously $0 < |\hat{\mathcal{C}}_t^{\max}| \leq n_{\text{honest}}$. Now we can formalize the impact of cheating failures on the connection graph.

Definition 7.2.1 (Impact of cheating failures)**.** *Let $\hat{\mathcal{C}}_t^{max}$ be the (weakly) connected component of maximum size in the subgraph G_t^{honest} of G_t induced by the set of honest nodes V_t^{honest}, at a given time t. We say that the protocol was subject to cheating iff there exist a sequence of points in time (of the size greater than one) such that $|\hat{\mathcal{C}}_t^{max}|$ and the number of edges in G_t^{honest} are strictly decreasing*[1]*.*

The above definition excludes a possibility of a **spontaneous partitioning** of the communication graph. It may happen during a fault-free execution of a given protocol due to its probabilistic nature and the limited knowledge about the network at each node (see Appendix C). However, in such case, the number of edges in G_t^{honest} would stay constant.

7.2.1.3 Simulation: the implementation and the execution scheme

In order to perform the analysis presented in this chapter, we implemented in Java a simulator for the protocol execution, including all cheating models presented in the following sections. The resulting framework contains a set of monitoring sensors able to track the complete state of the network — from the nodes to the individual cache entries. This allows gathering various network statistics (e.g. connectivity, clusters, etc.). Thanks to the GraphStream [46] Java library, the tool is able to graphically display the dynamics of the system (e.g. active connections, link updates, etc.).

[1] A sequence $(a_1, a_2, a_3, \ldots)$ is strictly decreasing iff $a_{i+1} < a_i$ for every $i \geq 1$.

During the execution, the whole simulation is divided into steps (called *simulation/synchronization steps*). In each of them, every node is selected once with a uniform probability to initiate a cache-exchange according to the protocol specification (see Section 7.1.2.1). That is: establishing an outgoing connection, sending own cache, receiving a cache from the destination and performing the cache merge.

At the onset of every experiment (step 0), the network is initialized as a bidirectional grid lattice (Figure 7.3a). Then we let the protocol run for 50 steps (i.e. **bootstrap**), which enables the network to converge into a self-organized equilibrium (Figure 7.3b). The actual time required to reach this desired state is usually below 25 steps and depends greatly on the actual configuration (the number of nodes, the cache size, etc.). The external interactions (all malicious activities) are made after the bootstrap, i.e. starting at the 51^{st} step.

All simulations were executed assuming idealised environment. This means that other faults can not occur. The nodes and links are fully operational throughout the whole simulation. In particular, making a connection from one node to another is always successful provided that the destination address exists.

7.2.2 Locating possible vulnerabilities through a data flow analysis

In order to better understand the root of the problem in the protocol design, we analysed what happens with each cache and all the entries during the execution. The experiments presented in this section are divided into two groups.
In the first group, the data flow characteristics were measured in a fault-free environment. This allows determining how the content of the caches are changing during the protocol execution. Additionally, it provides an insight about the spread of cache entries in the network, the time they are present after their creation and the number of connections made using them.
In the second group, the measurements were made for cheated entries. The analysis was conducted with two simplified fault models: a one-time and a constant corruption of the whole cache of a randomly selected node.

7.2.2.1 Data flow in a fault-free environment

In the first place, we wanted to check if the connectivity will spontaneously split without cheating faults present. For all network and cache sizes used in the experiments presented in this chapter, a connection graph was not divided for 100 executions, lasting 10000 simulation steps each.

Figures 7.4 and 7.5 present the amount of data exchanged between the nodes during a connection. This value is understood as the actual number of entries changed in the respective caches, not as the raw amount of data transferred between the nodes. As visible, the results are independent of the network size. The average number of entries changed for the cache size equal to 20 is around 12 (in the source cache) and 8 (in the destination cache) — respectively 23 and 14 for the cache size equal to 40. The spread of the results is maintained in both cases, independently on the network size. What might be surprising, is the lack of the symmetry between the entries changed at the source and at the destination. This indicates that, on average, the source of the connection has older entries in its cache than the destination.

Actually, this lack of symmetry is caused by the distribution of indegrees in the network (the outdegrees are always equal to the cache size). Figure 7.6 shows an example for a

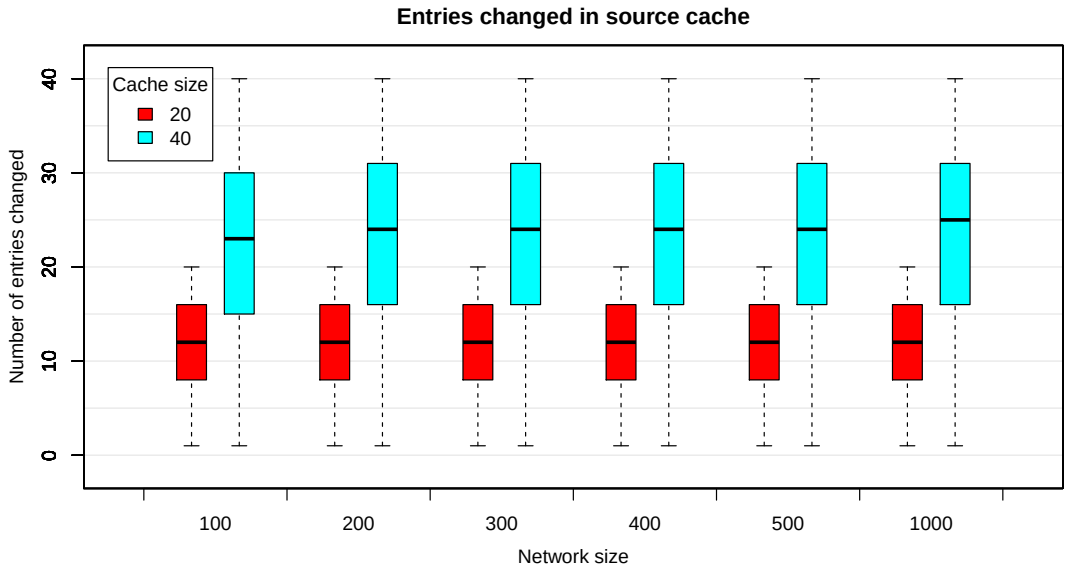


Figure 7.4: Number of entries changed during a single connection in the source cache. Values are measured over 100 executions (for the first 1000 merges) for all the parameter combinations.

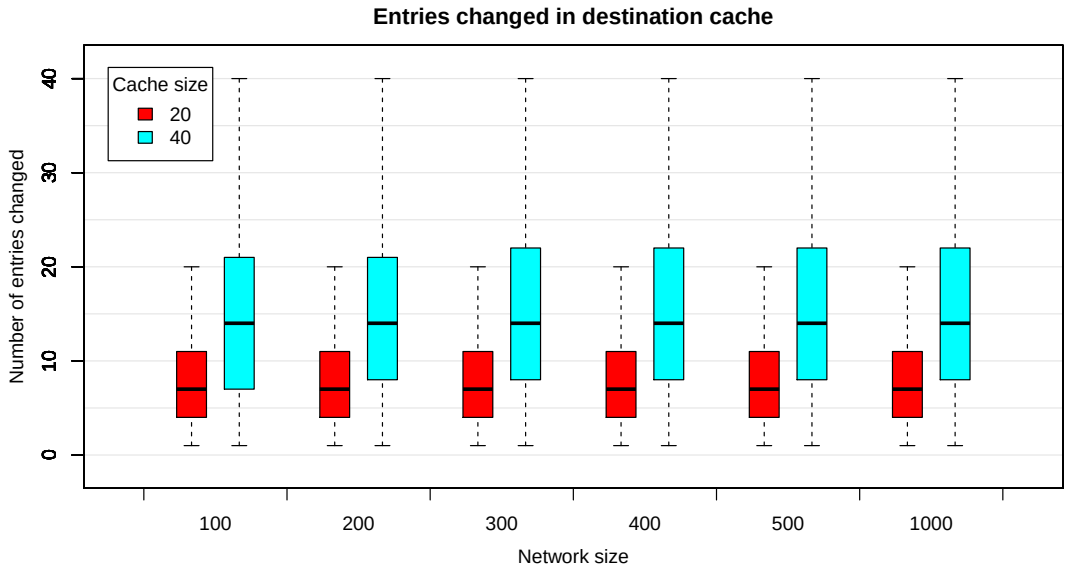


Figure 7.5: Number of entries changed during a single connection in the destination cache. Values are measured over 100 executions (for the first 1000 merges) for all the parameter combinations.

Newscast network consisting of 1000 nodes with the cache size set to 20. High values of the indegree are observed for more than half of the nodes, which makes them more probable to become a destination of some connection. Hence, causing more frequent cache merges and fresher content.

Figure 7.7 shows the number of nodes visited (inserted into the cache of a node) by the cache entries, Figure 7.8 — the time spent in the network, and Figure 7.9 — the number

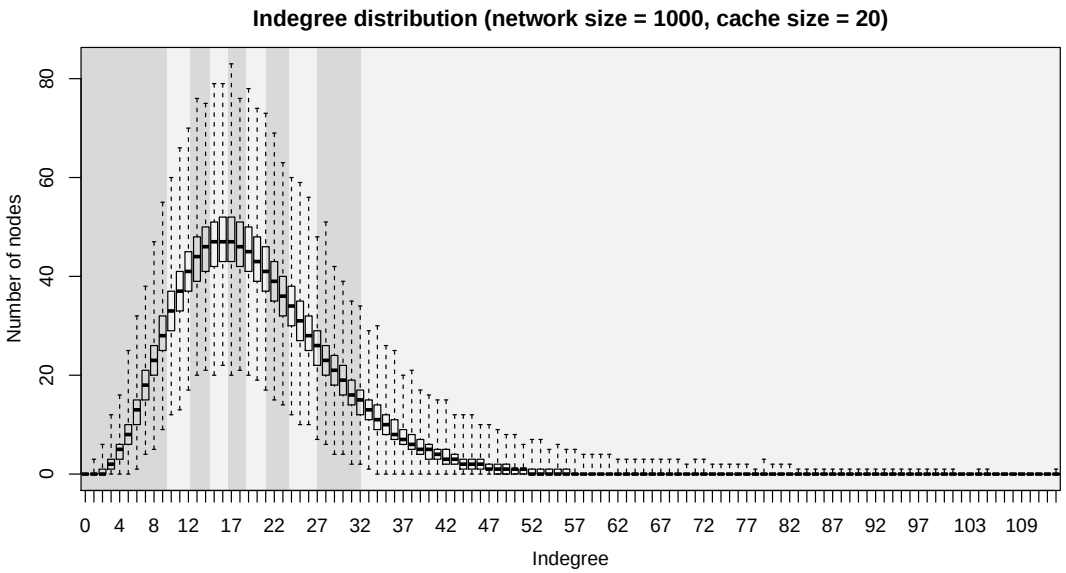


Figure 7.6: Example of the indegree distribution for a network consisting of 1000 nodes with the cache size set to 20. The values are measured at the beginning of the simulation step for 1000 steps (after bootstrap) for 100 executions. Altered colors in the background correspond to the number of nodes equal to 10% of the original network.

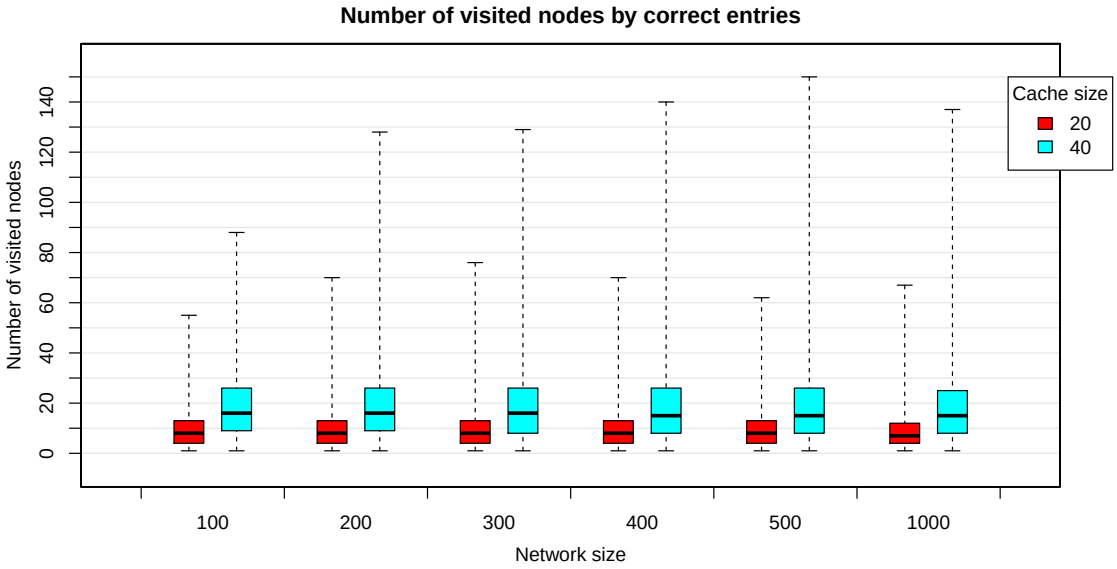


Figure 7.7: Number of visited nodes measured for the first 10000 correct entries, fully removed from the network, for 100 executions for all the parameter combinations.

of connections established using the entries present in the network. Whiskers on the figures mark the extremes. As visible, all measured values are on average independent from the network size and depend only on the cache size. For small networks, some cache entries may spread to most of the nodes in the network (for instance, the maximum value obtained for the network size equal to 100). However on average they reach 9 (respectively 19) nodes —

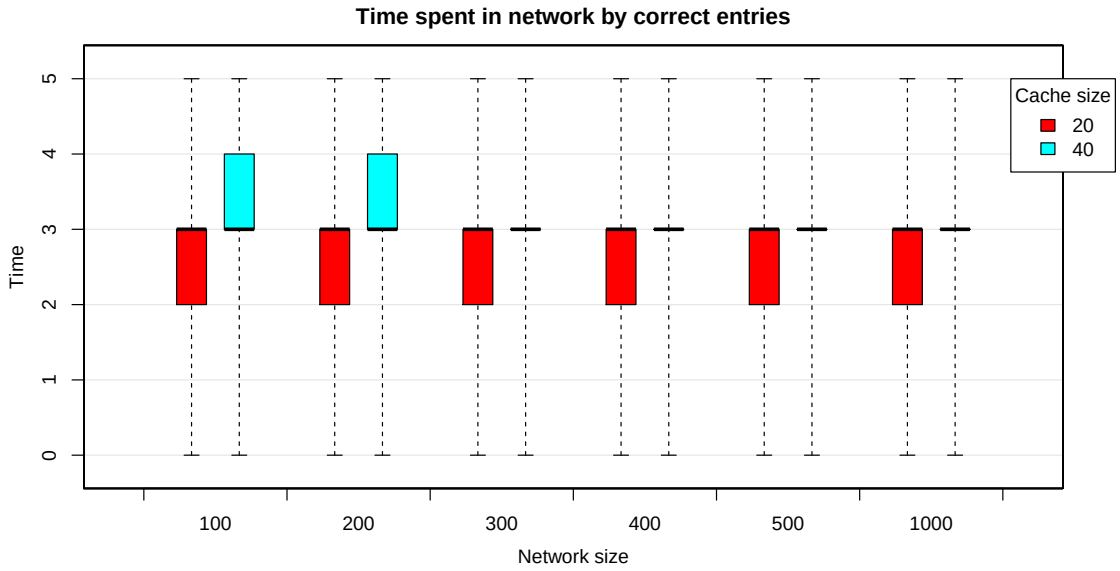


Figure 7.8: Time spent in the network by the first 10000 correct entries, fully removed from all the nodes, for 100 executions for all the parameter combinations.

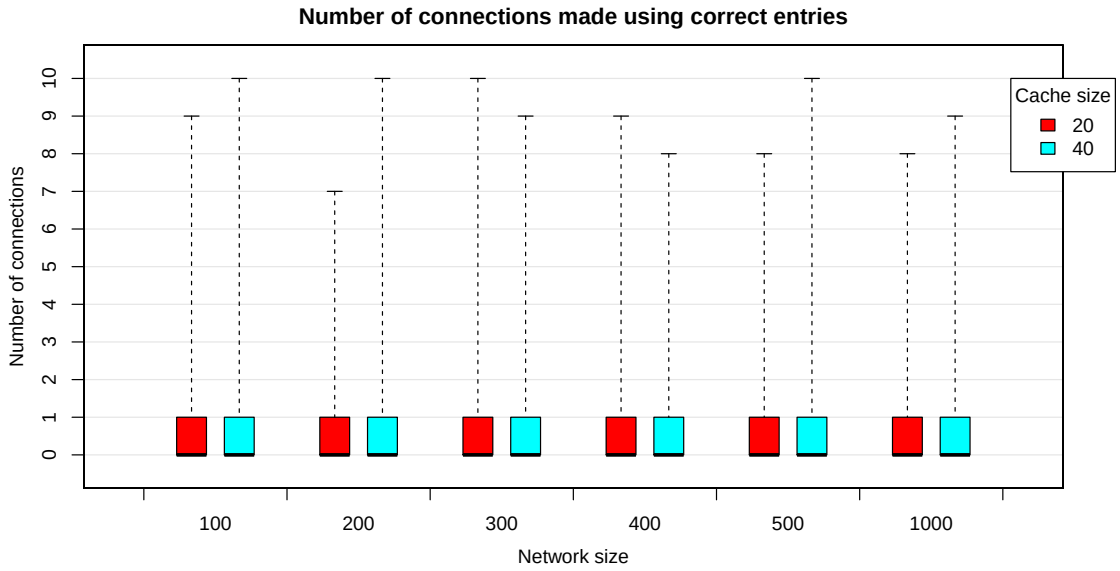


Figure 7.9: Number of connections made using the first 10000 correct entries, fully removed from the network, for 100 executions for all the parameter combinations.

for the cache size equal to 20 (respectively 40). The mean number of synchronization steps required to completely remove an entry is equal to 3 for all the test cases, and in the worst case — 5. Majority of the cache entries created during the protocol execution are never used to make an outgoing connection, where at most some entries are used only 10 times.

7.2.2.2 Flow of cheated entries

Turning now to the analysis of the cheated entries in the network subjected to the influence of simple cheating models. First, the altered items are injected once into the cache of a node

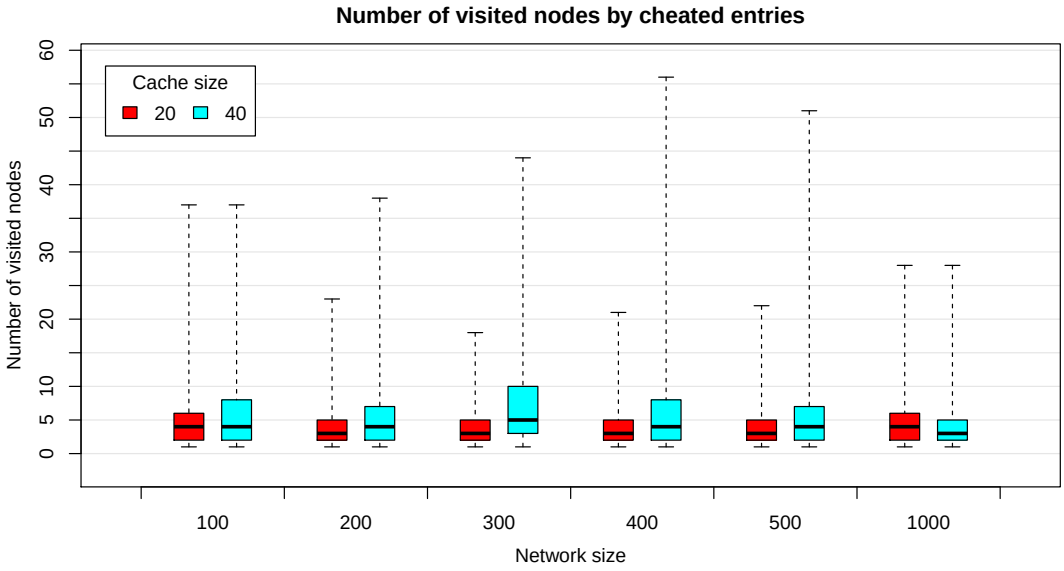


Figure 7.10: Number of visited nodes measured for cheated entries, fully removed from the network, for 100 executions for all the parameter combinations. Cache size of them are injected into the cache of a randomly chosen node at the random moment during the 51st synchronization step (i.e. after bootstrap).

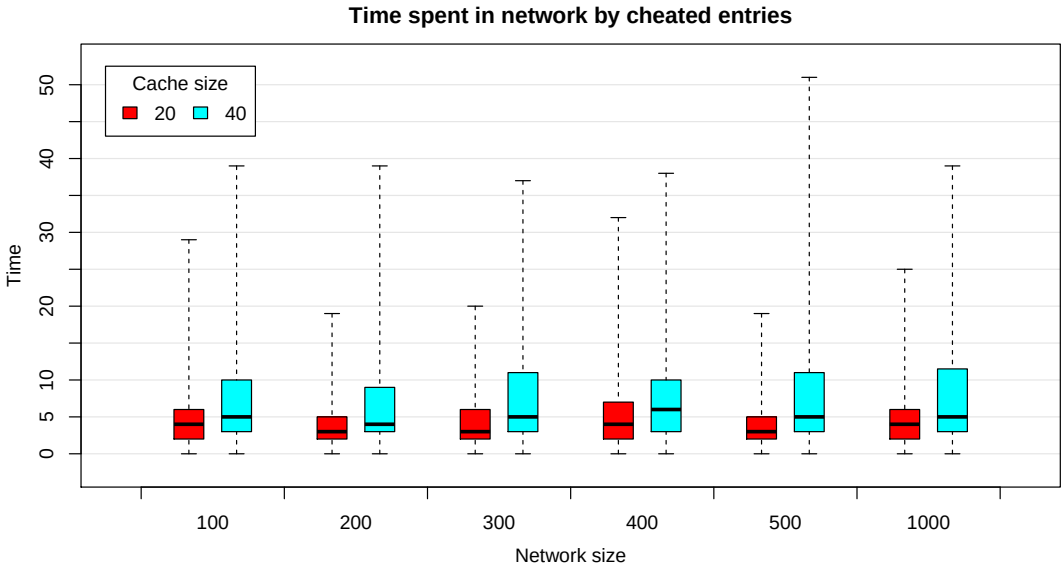


Figure 7.11: Time spent in the network by cheated entries, fully removed from all the nodes, for 100 executions for all the parameter combinations. Cache size of them are injected into the cache of a randomly chosen node at the random moment during the 51st synchronization step (i.e. after bootstrap).

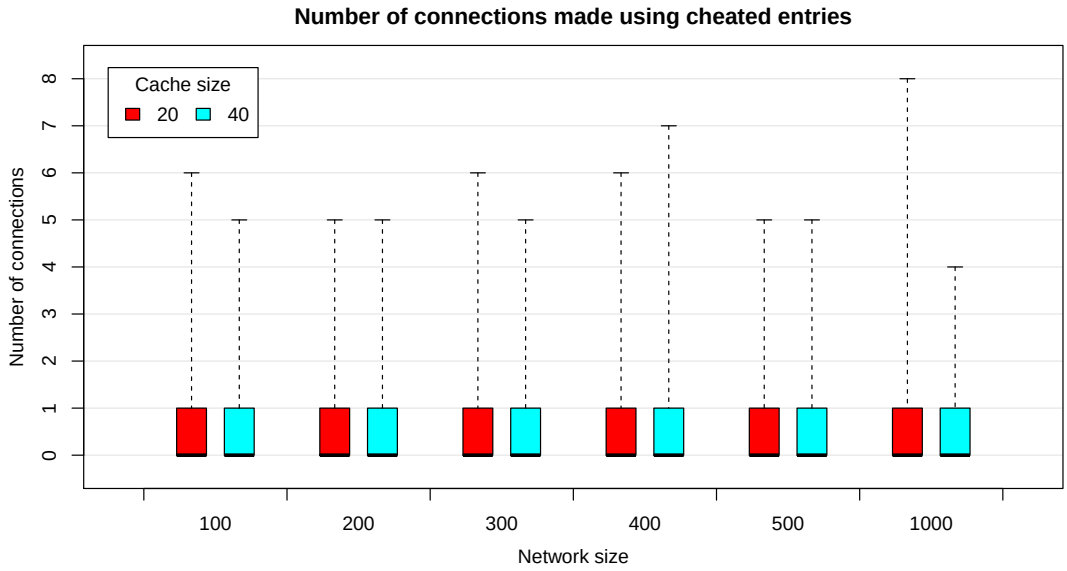


Figure 7.12: Number of connections made using cheated entries, fully removed from the network, for 100 executions for all the parameter combinations. Cache size of them are injected into the cache of a randomly chosen node at the random moment during the 51st synchronization step (i.e. after bootstrap).

chosen accordingly to a uniform distribution law, at the random moment (again selected uniformly), during the first synchronisation step after bootstrap. Results are depicted in Figures 7.10, 7.11 and 7.12.
As immediately visible, the information spread in the network (Figure 7.10) is worse than previously — maximum values are below half of the ones obtained earlier and the averages are below 5 nodes. It is caused by the simplified cheating model. The whole cache of the chosen node is corrupted, hence it is not able to successfully initiate any outgoing connection. This situation persists until the first successful merge operation is invoked by some other member of the network, which in some rare cases may never happen as a consequence of the indegree distribution (see Figure 7.6).
The results presented on Figure 7.11 show the self-healing property of Newscast in case of cheating failures. The cheated items are removed on average in around 5 synchronisation cycles, independently on the network size. If a single node would have to seed out the corrupted data without help from other nodes, it would take at least "cache size" synchronisation cycles. Interestingly, an extreme case was observed for the network consisting of 500 peers and the cache size equal to 40. It took more than 50 simulation steps to remove the cheated entries. The situation was caused by their rapid spread in the neighbourhood of the attacked node which resulted in swapping the cheated entries between a group of nodes.
Similar to the fault-free environment, most of the corrupted entries are never used to initiate an outgoing connection — a failure causes the removal of the entry from the cache. Therefore, in most cases the cheated data is present in the network until its replacement with fresher information.

Figures 7.13 and 7.14 presents results obtained when the content of the cache of a randomly chosen node is flooded with cheated entries at every simulation step. The time required to disconnect any node is surprisingly short, since it averages on 8 simulation steps for cache size

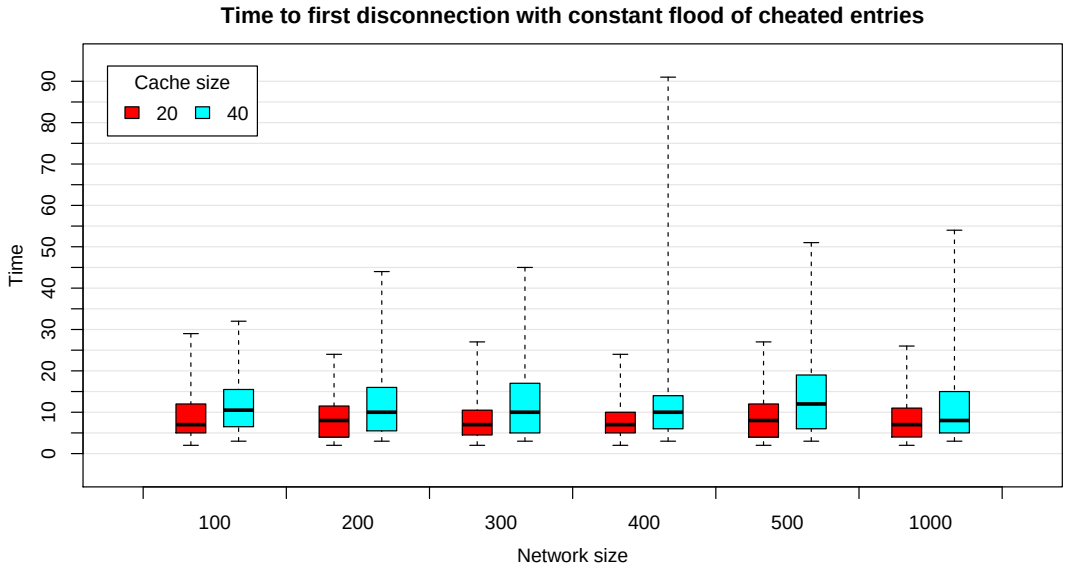


Figure 7.13: Time to the first disconnection in the network with the cache of a randomly chosen node (fixed during the execution) constantly flooded with cheated entries. The data was gathered for all the combinations of simulation parameters over 100 executions.

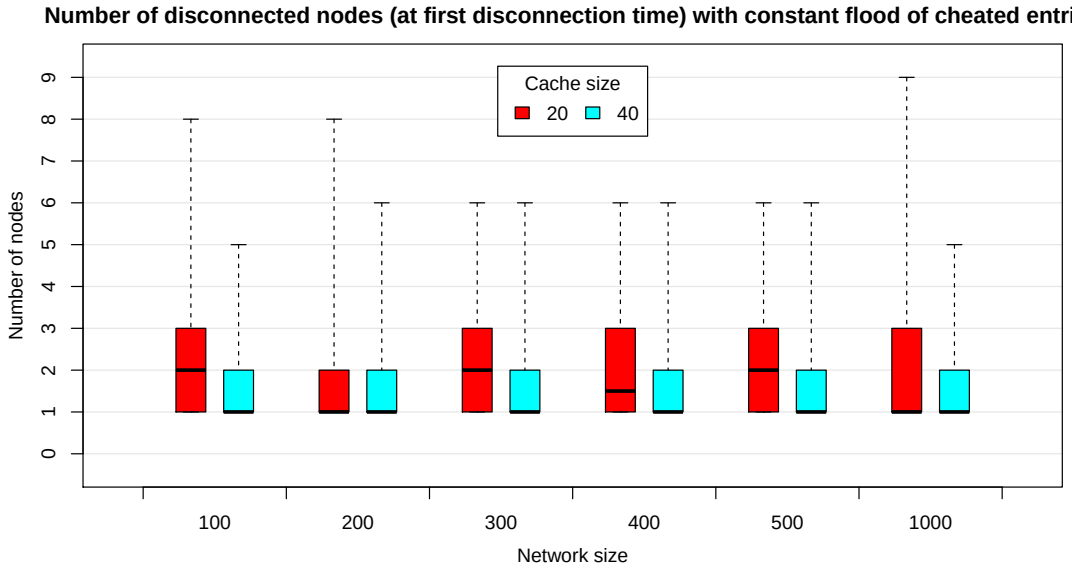


Figure 7.14: The number of disconnected nodes (at the first disconnection time) with the cache of a randomly chosen node (fixed during the execution) constantly flooded with cheated entries. The data was gathered for all the combinations of simulation parameters over 100 executions.

equal to 20 (respectively, 10 for the cache size equal to 40), for all the network sizes. In some extreme cases even 90 synchronization steps are needed (see result obtained for the network consisting of 400 nodes with cache size set to 40). It is worth to notice that not always the flooded node is disconnected as the first and only one. Average number of nodes affected by the cheating during the experiments oscillates between one and two. The maximum value obtained — 9 — occurred for a network consisting of 1000 nodes with cache size set to 20. For every single execution, at least one node got disconnected.

7.2.2.3 Summary of the results

During the execution of Newscast, indegrees of the nodes are distributed unevenly. However, the observed values of this statistic are changing over time. For instance, a node having the highest indegree might have the lowest number of neighbours after few synchronisation cycles. Such imbalance in the indegree distribution promotes better spread of the information from a group of the nodes to the rest of the network. The peers having many neighbours posses (and quickly acquire) the freshest cache entries from their surrounding.

On average, an entry created at a given node spreads to almost half of the cache size of its neighbours within three synchronisation cycles. Additionally, the address which it contains is hardly ever used to establish a connection. Hence, they are almost never verified.

A naive approach to corrupt with cheated entries the whole cache of a given node is not very effective. This stops partly the spread of the malicious data in the network, affecting fully the attacked peer and slightly some of its neighbours. The Newscast network is able to sift the cheated entries (i.e. self-heal), independently if the corruption was performed once or constantly, in the worst case loosing a very limited number of nodes.

In the next section, we will design and analyse for a malicious client of Newscast. Basing on the results presented above, we will propose an effective scheme for independent, non-cooperative cheaters.

7.2.3 The connectivity-splitting attack

As discussed earlier, a naive malicious approach involving the corruption of the whole cache of a given node is not very effective. However, an option to implement an evil-intentioned client of the protocol was unexplored. In this section we will propose and analyse such solution, basing on the discoveries made previously.

7.2.3.1 An effective scheme for a malicious client

Implementing from scratch a malicious client of the protocol opens more possibilities compared with altering the already implemented scheme. Such evil-intentioned peer does not have to follow any of the previously imposed constraints. Most of all, it can actively choose the nodes as potential victims of the attack. Further, its view does not have to be limited by the cache size, as the cheater may gather information about the global state of the network during the protocol execution.

When a cheating client makes or responds to a connection, it sends a cache filled in with cheated entries (the freshest possible, containing random addresses — as explained earlier in Section 7.2.1.1). Nodes which receive and merge such data would have (with a high probability) only one valid address (referring to the cheater) to make an outgoing connection. However, there is a high chance that some other peer contains an entry pointing to the victim

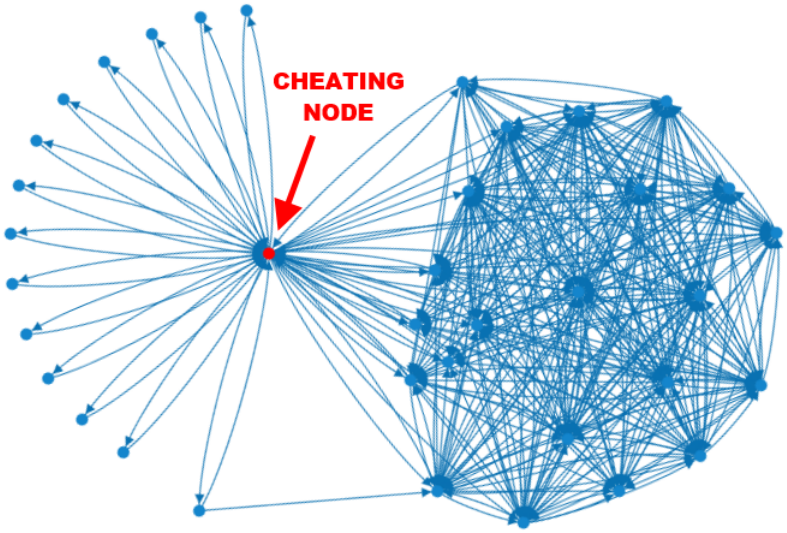


Figure 7.15: A visualisation of a problem with cheaters accepting addresses from incoming connections. Nodes connected only to the cheater start to flood him with random addresses.

node. Therefore, after some time, the corrupted cache may be restored to a partially valid state by an incoming connection (i.e. self-heal). Nevertheless, there is a side effect of this process, as the falsified information will also spread virally into the network. This raises questions which will be answered by the experiments:

1. How many times does the cheater have to connect to a node to fully disconnect it?
2. Do these connections have to be consecutive?

Consequently, we have also explored the option for cheaters to choose more than one target at a time, following the round-robin principal. The victims are exchanged after the desired number of connections to them is reached (a **frequency of target change**).

Each cheater inserts independently the cache size c of random addresses into the network at each cache-exchange. Over time, some generated entries are returned to them. This problem is most noticeable after a significant number of nodes are connected only to the malicious client (see Figure 7.15). Thus, we propose to ignore information received during the incoming connections. This will only slightly affect the process of discovering the network, because information about its members is still obtained through the outgoing communication.

What is more, due to the lack of the cooperation, malicious nodes do not know about each other. Consequently, the information about which addresses where generated by the other malicious nodes is not shared. Moreover, cheaters could connect to each other, flooding themselves with random entries. This could lead to a situation when the set of discovered addresses is very big, mainly filled by the invalid entries. As a solution, we propose to assign a priority for discovered addresses, determining the target of the next outgoing connection. Its value is defined by the number of occurrences of a given address during all the previous cache-exchanges. It is motivated by the fact that the valid entries should appear more often than the generated ones. Additionally, it allows following the observed indegree distribution, improving the spread of cheated entries in the network. After a successful connection to a given address, its priority is decreased by one to ensure that the cheater will not be stuck with a single target.

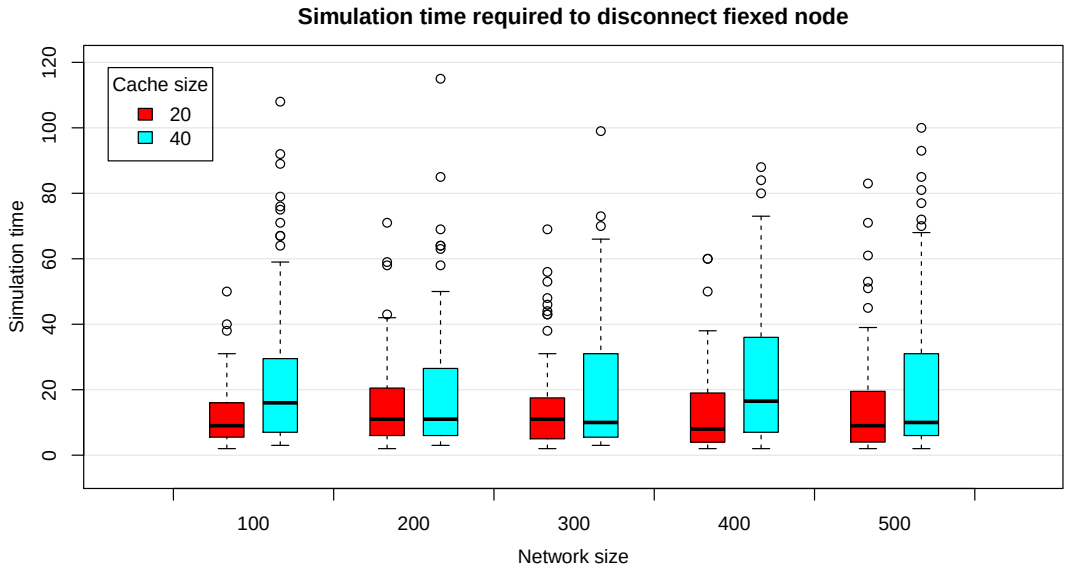


Figure 7.16: Time required to disconnect a single, uniformly selected at random node.

7.2.3.2 Discovering the optimal parameters for a malicious client

The purpose of the first experiment was to determine an estimate on the simulation steps required to completely disconnect one peer from the system. Figure 7.16 shows the results for a one cheater attacking a single, uniformly selected at random target from the network. As is visible, the obtained values are comparable within the same cache size settings, independently from the network size. In most of the cases, the simulation time required to remove all links to and from a single target fits in the range from 1 to 20. Basing on these results we have chosen a range for frequency of target changes from 1 to 20 for the upcoming group of tests. It is also worth to note here, that the persistent attack on a single node can take a long time — as it is required for the network to lose all the information about the target. In some cases it took above 100 simulation steps to complete the task by the cheater. This time mainly depends on how well the given address is widespread in the network, as how it was explained earlier — any incoming connection can partially heal the cache of the node.

The last group of the experiments required to configure the malicious client was aimed to determine how frequently the target has to be changed to obtain the best performance. Additionally, it provided the data helpful to decide whether it is beneficial to attack at once more than one node. Figures 7.17 and 7.18 present the results. As is clearly visible, the best performance is obtained with one target changed for each outgoing connection. Other combinations of cheater parameters are causing degradation of efficiency and stability of the performance. In the optimal configuration, the network is fully disconnected in less than 800 simulation steps (on average) for a cache size equal to 20 and in less than 1000 simulation steps for a cache size equal to 40 (values are better visible on Figure 7.19). Results for smaller networks are not presented, as they follow a similar trend (only the scale is different).

7.2.3.3 Assessing the performance of the attack

Finally, having the full configuration of a non-cooperative malicious client, we move to the assessment of their efficiency. Figure 7.19 presents the average speed of the network degra-

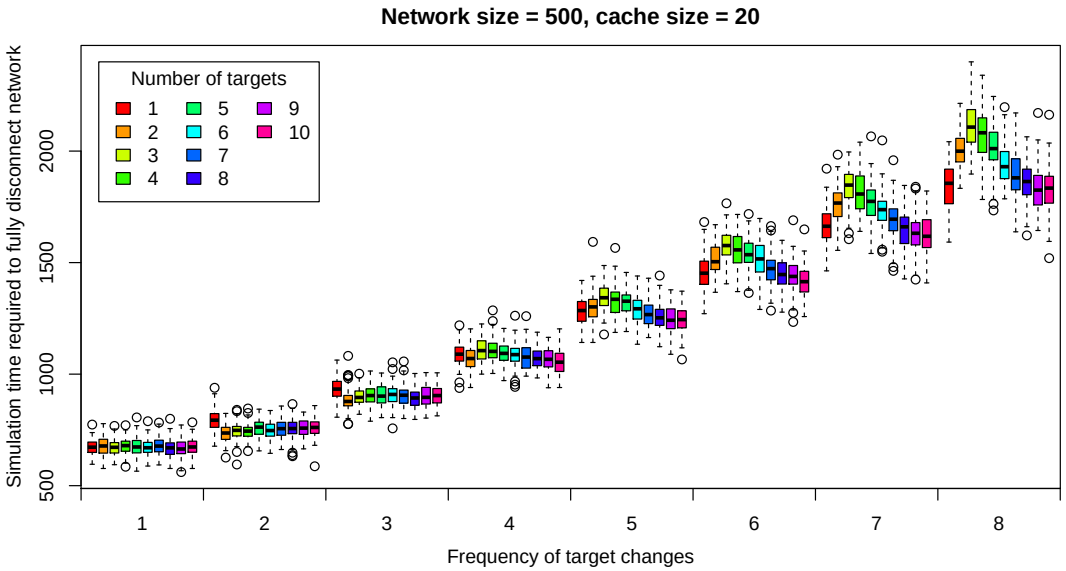


Figure 7.17: Influence of frequency of target changes and number of targets on network disconnecting time with one cheater present in the network consisting of 500 nodes and cache size equal to 20.

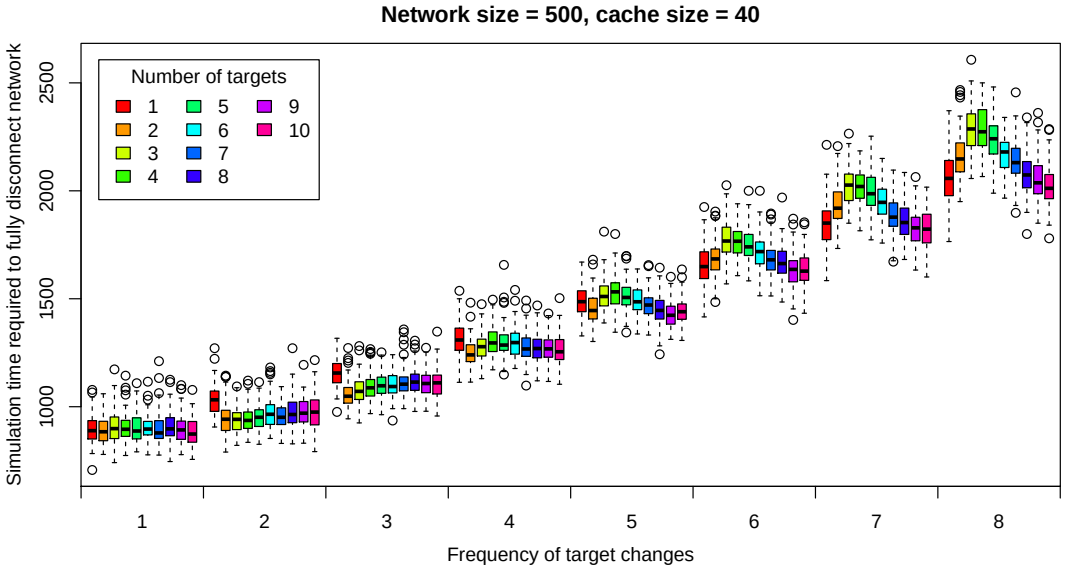


Figure 7.18: Influence of frequency of target changes and number of targets on network disconnecting time with one cheater present in the network consisting of 500 nodes and cache size equal to 40.

dation for different numbers of nodes and both cache size settings. All the values are scaled to the range of $[0, 1]$ through the division of the average biggest cluster size (from 100 executions) by the network size, i.e. $|\hat{\mathcal{C}}_t^{\max}|/|V_t^{\text{honest}}|$ (which yields the probability that a given node $n \in \hat{\mathcal{C}}_t^{\max}$). The speed is almost linear for all the combinations of the relevant parameters, until there is a small cluster left (consisting of less than 20% of the original number of nodes). Nevertheless, the network is fully disconnected in less than 1000 simulation steps for

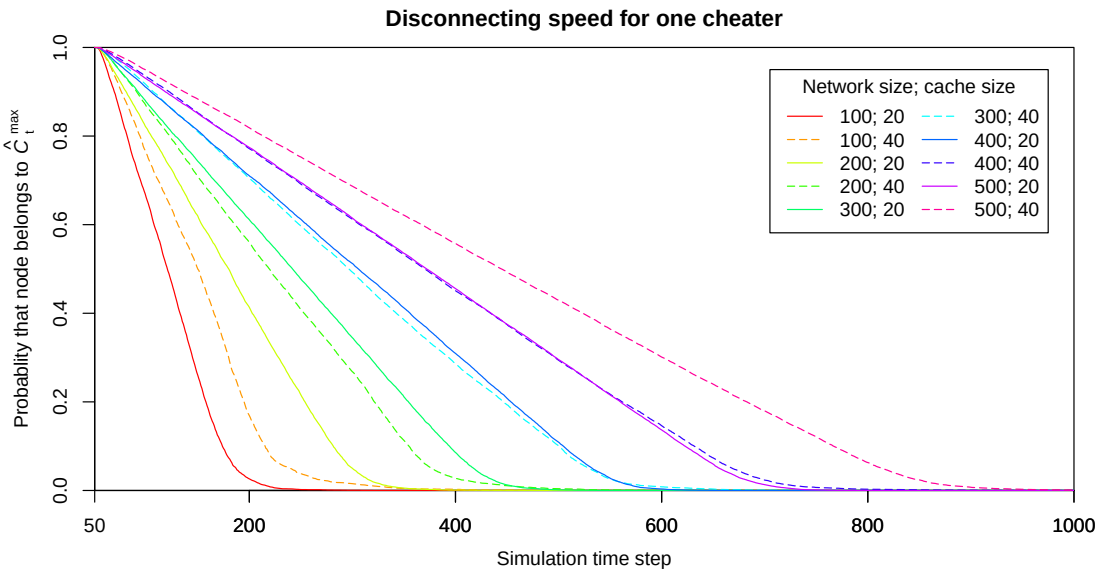


Figure 7.19: Disconnecting speed compared between different network and cache sizes with only one cheater. Probability that a node belongs to the $\hat{\mathcal{C}}_t^{\max}$ is computed by division of the average biggest component size by the network size.

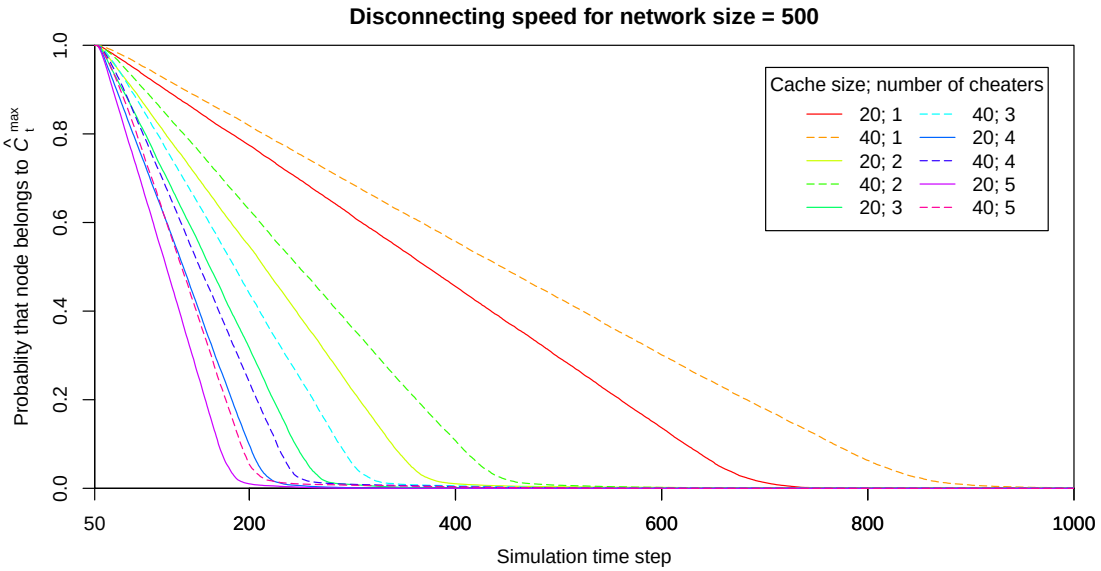


Figure 7.20: Disconnecting speed compared between different cache sizes and amount of cheaters working in the network consisting of 500 nodes. Probability that a node belongs to the $\hat{\mathcal{C}}_t^{\max}$ is computed by division of the average biggest component size by the network size.

all the cases. Given that the actual Newscast implementation[2] sets the synchronisation time to 10 seconds, 1000 simulation steps stand for less than three hours of a real execution of a P2P system. This allows us to conclude that even one cheater in the network, following the presented model, can have a devastating effect on the functioning of the Newscast network.

[2] `http://dr-ea-m.sourceforge.net/`

So far, we have shown and described results for only one cheater attacking the protocol. Figure 7.20 presents the influence of the number of cheaters varying from 1 to 5 on the network consisting of 500 nodes with cache sizes set to 20 and 40. It is important to emphasize here again, that cheaters do not know about each other and do not collaborate. Despite the lack of any kind of information sharing between malicious nodes, more of them causes faster loss of the connectivity — for 5 malicious nodes it happens in less than 300 simulation steps, which corresponds to one hour of a real execution time (as mentioned above). Additionally, it is visible that the influence of the bigger cache size loses its value when more cheaters are present in the network (the gap between both settings decreases).

Finally, Figures 7.21 and 7.22 present the scalability of the solution. If the number of cheaters scales proportionally with the network size, the efficiency of the malicious nodes can be maintained. Moreover, cheating nodes do not interfere with each other, which is the effect of prioritizing the choice of a connection destination according to its frequency of appearance during the interaction with the rest of the network.

7.2.3.4 Summary of the results

The proposed scheme for a malicious client proved to be effective and scalable. Prioritising the targets by their frequency of appearance during the cache-exchanges is the only mechanism required for the effectiveness of the solution. The scalability is achieved thanks to ignoring the cache entries from incoming connections. With such settings, cheaters do not disturb each other much and do not get stuck on cheating one another.

The analysis shows that the uncoordinated, independent cheaters are a serious threat for Newscast. Even a few malicious clients can break the connectivity in the network really fast. Moreover, the model is hard to detect at the level of each node as it does not lead to the formation of any anomalies easily detectable in the neighbourhood. The only symptom is a decreasing number of (working) links between the hones nodes, which is similar to churn. In case when the random addresses can not be used, they might be substituted with the ones coming from the discovered set. The least represented nodes during the cache-exchanges are most likely inactive.

7.2.4 Countering the attack

So far this chapter focused on a way to break the connectivity in the Newscast network, exploiting the protocol design. In the following section we will propose an improvement to the execution scheme allowing tolerating uncoordinated malicious clients.

Simplicity of Newscast aids exploring various scenarios and extensions. The application considered in this thesis — the distributed execution of EAs and namely Evolvable Agent Model (EvAg) — allows relaxing some requirements in contrast to the typical approaches found in the literature (see Section 7.1.2.2). As discussed earlier in Section 5.3.2.1, some computing resources may be lost during the execution of the algorithm without diminishing its performance. Therefore, the absolute resilience against cheaters at all times is not needed — the fluctuations in the number of nodes participating in the computation may be present.

A typical DGVCS has two main types of machines: belonging to the owner of the platform and volunteered (see Section 3.1.3). The first type is characterised by its high stability, security and a full control. On the contrary to the second, consisting of volatile resources with the lack of security guarantees and control.

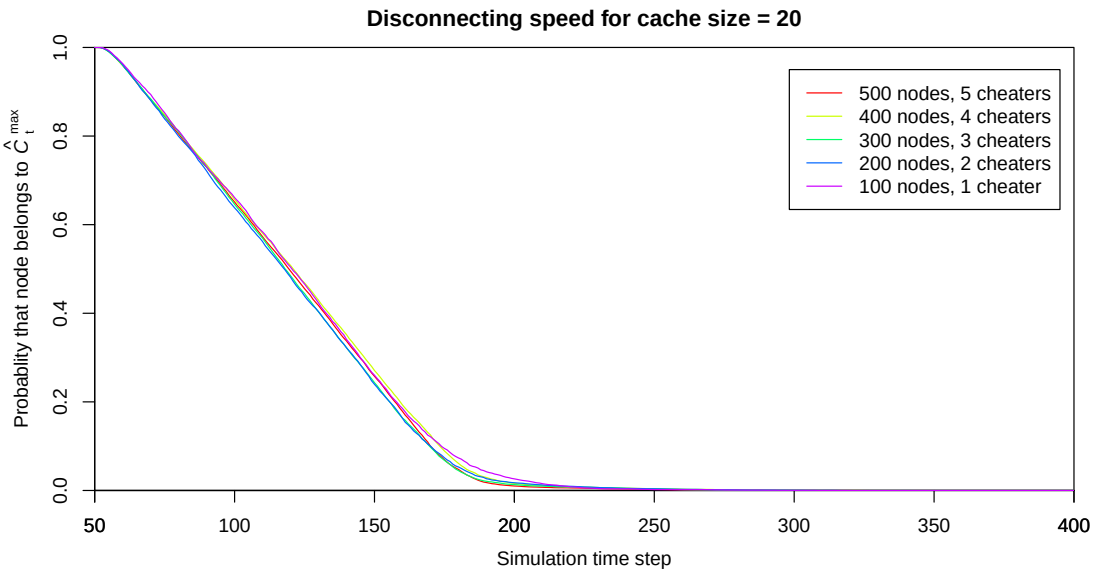


Figure 7.21: Scalability of the proposed malicious client for different network sizes and the cache size equal to 20. Probability that a node belongs to the $\hat{\mathcal{C}}_t^{\max}$ is computed by division of the average biggest component size by the network size.

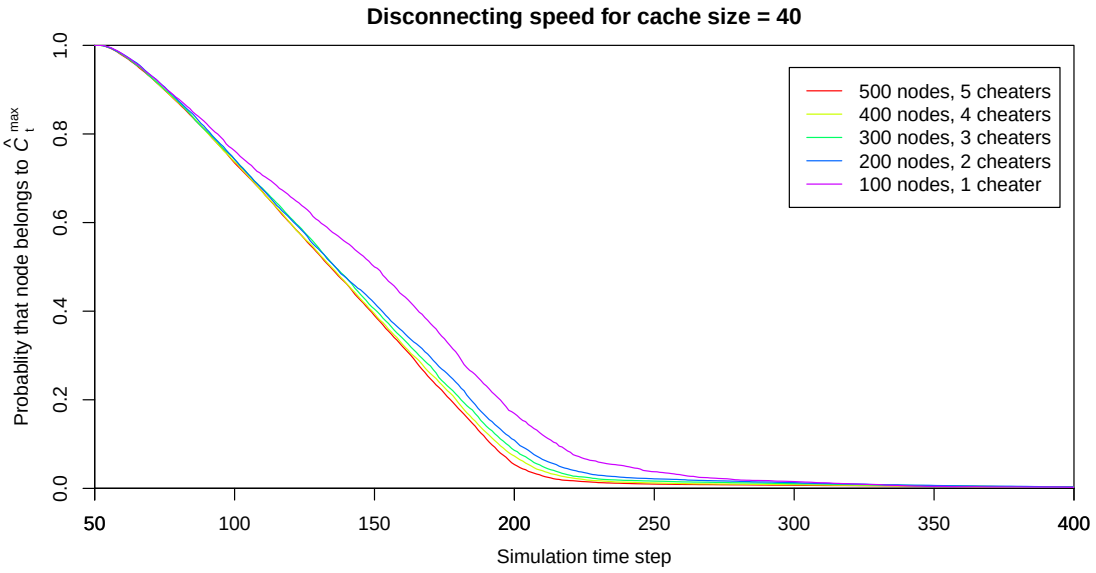


Figure 7.22: Scalability of the proposed malicious client for different network sizes and the cache size equal to 40. Probability that a node belongs to the $\hat{\mathcal{C}}_t^{\max}$ is computed by division of the average biggest component size by the network size.

We propose to delegate some machines belonging to the DGVCS owner to participate in the computation. Only these nodes should be used as the entry points for new volunteers joining the system. The entry point can be obtained from the central server managing the whole computation. The delegated machines might share with each other the set of active volunteer addresses. An encountered client which did not join the computation through any of the entry points might be easily blocked. Any address outside of the registered pool can be sift from the cache entries flowing through the nodes acting as the entry points. Moreover,

these delegated resources can be utilised to gather and verify the results.

However, such approach alone can not guarantee maintaining the connectivity between the nodes. The volunteered resources should contact some entry point from time to time. During such connection, the whole cache of the peer should be overwritten with the received entries. Such approach corresponds to the reconnecting to the network, but without restarting the computation at the given node. The periodicity of the reconnection is the most important aspect. Too low values would cause forming the communication hubs at the entry points, increasing their load required to serve all the requests. What is more, the self-organisation property of the network may be lost. Hence, the reconnection period should be as high as possible.

The original design, as shown before, may lose the connectivity really fast. It would require a lot of designated machines to distribute the load caused by the low reconnection periods. However, the resilience to cheating faults could be increased by limiting the number of exchanged cache entries during a connection (without modifying the cache size). This in turn would decrease the impact of cheaters on the connection graph and the spread of corrupted data in the network. Therefore, we will analyse in this section the influence of this limit on the connection graph and its effectiveness against the previously proposed malicious client.

7.2.4.1 Limiting the merge against uncoordinated malicious clients

The approach to limit the merge operation is motivated by the fact, that plain tampering with the time-stamps of the cache entries allows them to spread rapidly in the network. As cheating clients actively initiate the outgoing connections, they can choose a specific node and overwrite its cache — all $c \in \mathbb{N}$ entries. The same happens when any of the cheaters responds to the incoming request. However, during the connection between non-malicious peers, the number of updated items is a lot lower. In case of the destination of the connection, the change in the view does not exceed (on average) half of the cache size; in the source — its only slightly above the mentioned value.

As visible on Figures 7.23 and 7.24, limiting the number of exchanged cache entries to at least $c/2$ of the freshest items rapidly decreases the effectiveness of a single cheater. Within 1000 simulation steps, only few nodes are disconnected, independently from the cache size. We observed the same trend for smaller networks (results are omitted for clarity). The 40% limit (8 and 16 items for the respective cache size equal to 20 and 40) allows maintaining almost full connectivity throughout the execution.

With the increased number of malicious clients (to 10) the situation changes. Figures 7.25 and 7.26 present the results. The performance of cheating is still impaired, but less effectively. In this case there is a bigger difference between the results obtained for different cache sizes. Without any limitation on the cache merge, the networks with the cache sizes equal to 20 were fully disconnected in around 50 simulation steps faster than for the same setups with the bigger views. However, as soon as the limitation on the merge operation is introduced — the difference in cheating-tolerance is more apparent. For a small cache and the 50% limit, less than 10% of the nodes stay connected in the biggest component at the end of the simulation. For a cache size equal to 40, more than 50% of the peers can still cooperate.
In the previously optimal configuration — the 40% limit (8 and 16 items for the respective cache size equal to 20 and 40), some nodes are still lost. For the view of the size 20, more than 20% of nodes are disconnected, for 40 — more than 10%. Decreasing the limit further,

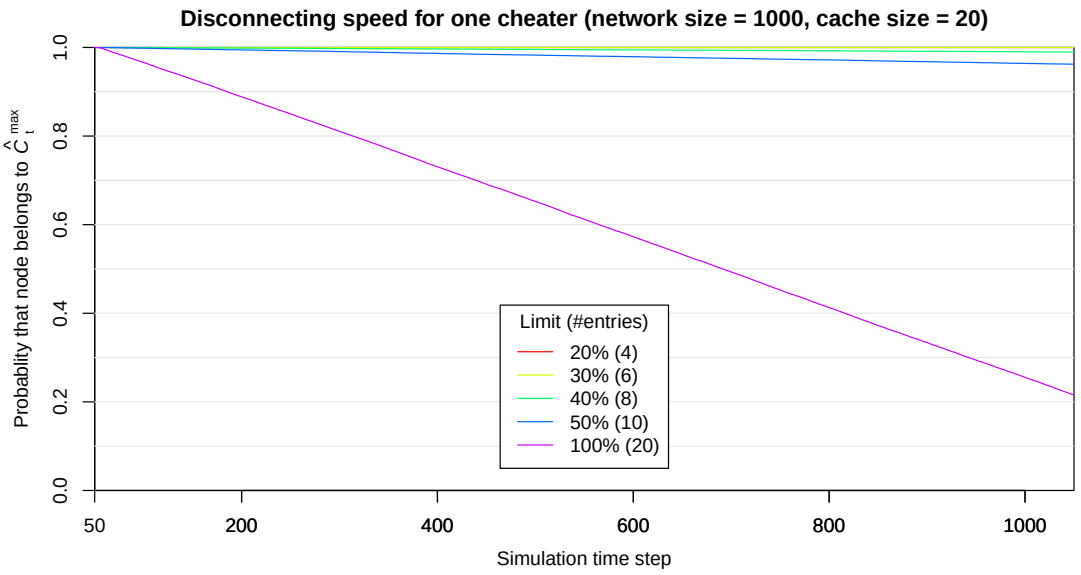


Figure 7.23: Disconnecting speed for one cheater, compared for different limits on the cache exchange for the Newscast network consisting of 1000 nodes with the cache containing 20 entries.

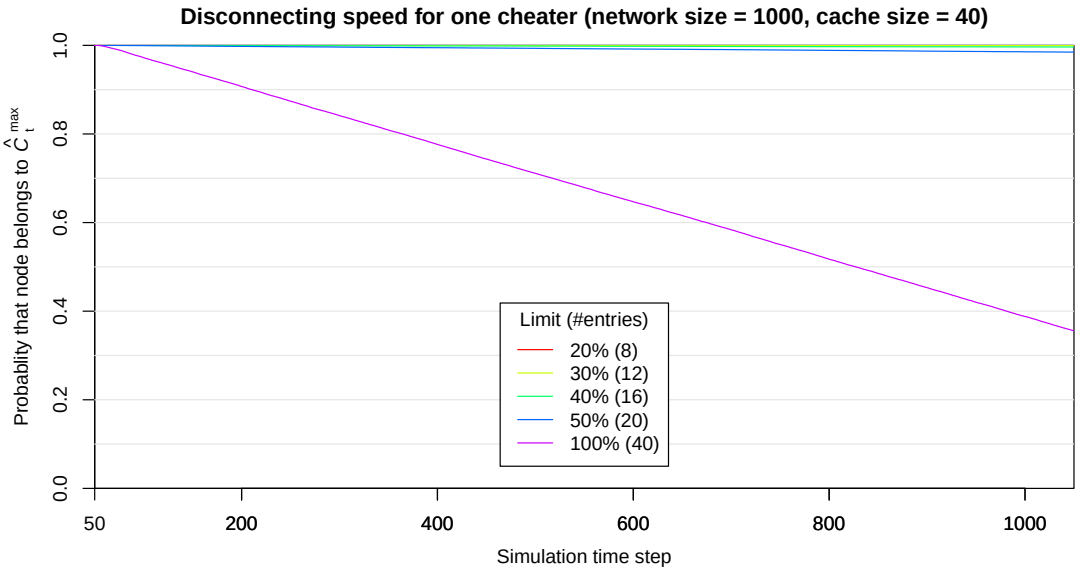


Figure 7.24: Disconnecting speed for one cheater, compared for different limits on the cache exchange for the Newscast network consisting of 1000 nodes with the cache containing 40 entries.

solves the problem. Thus, if the merge operation is limited to the 30% of all the entries in the cache, the network stays almost fully connected and the end of the execution.

Further increase in the malicious activity (to 100 cheaters) can be matched again with an appropriate limit. Figures 7.27 and 7.28 present the results. Limiting cache-exchange operation to 15% of the freshest entries allows tolerating the cheaters in the network with a slight loss of the computing resources (less than 10% at the end of the execution). It is worth noting that the increasing number of the malicious clients combined with the decreasing limit

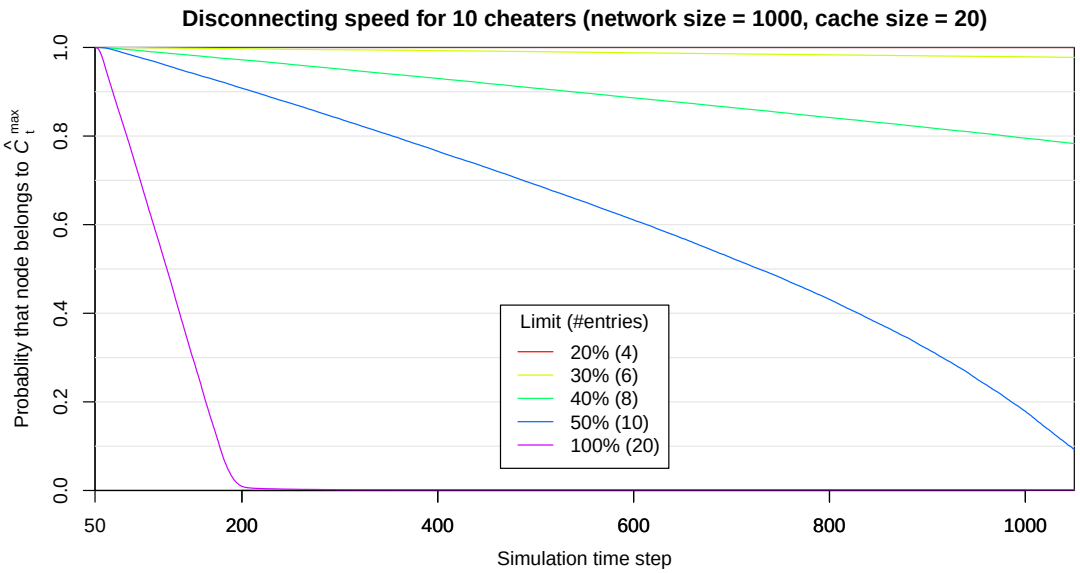


Figure 7.25: Disconnecting speed for 10 cheaters, compared for different limits on the cache exchange for the Newscast network consisting of 1000 nodes with the cache containing 20 entries.

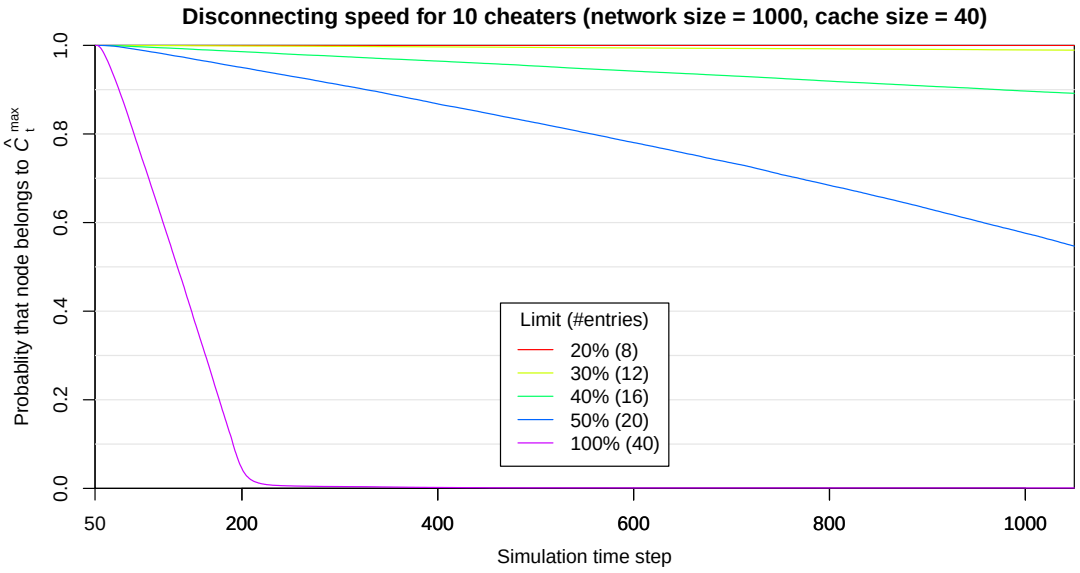


Figure 7.26: Disconnecting speed for 10 cheaters, compared for different limits on the cache exchange for the Newscast network consisting of 1000 nodes with the cache containing 40 entries.

starts to reverse the benefit from the bigger cache size. The influence of cheaters is greater for the networks consisting of nodes with bigger views (compare Figures 7.25 and 7.27 with 7.26 and 7.28).

In conclusion, the increasing number of malicious clients in the network may be matched by the bigger cache sizes combined with the limitation on the exchanged items. The complete solution depends on the expected number of cheating participants. In the optimal case, the approach presented in this section should be combined with a detection method of the

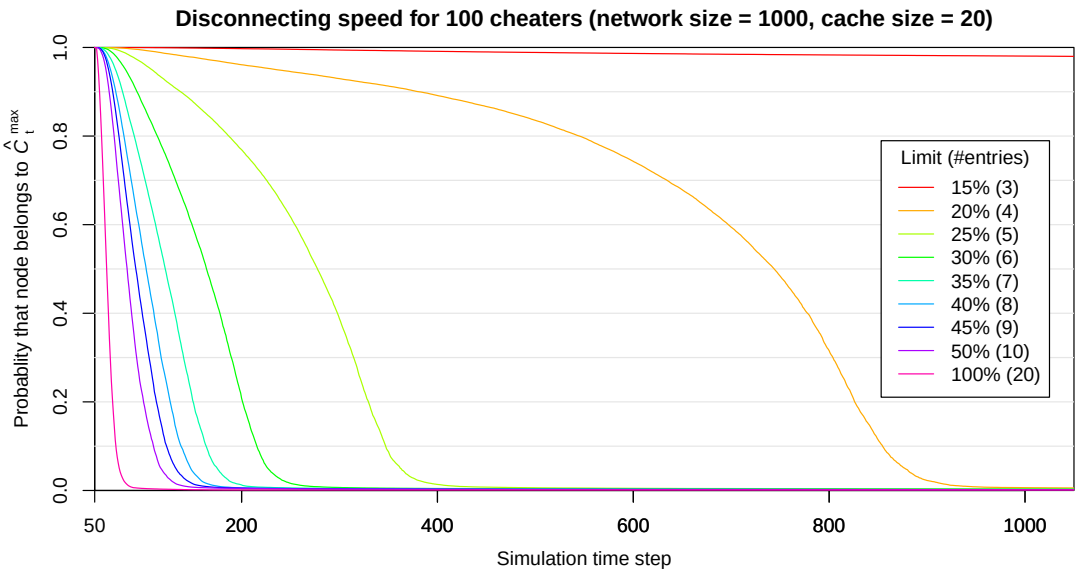


Figure 7.27: Disconnecting speed for 100 cheaters, compared for different limits on the cache exchange for the Newscast network consisting of 1000 nodes with the cache containing 20 entries.

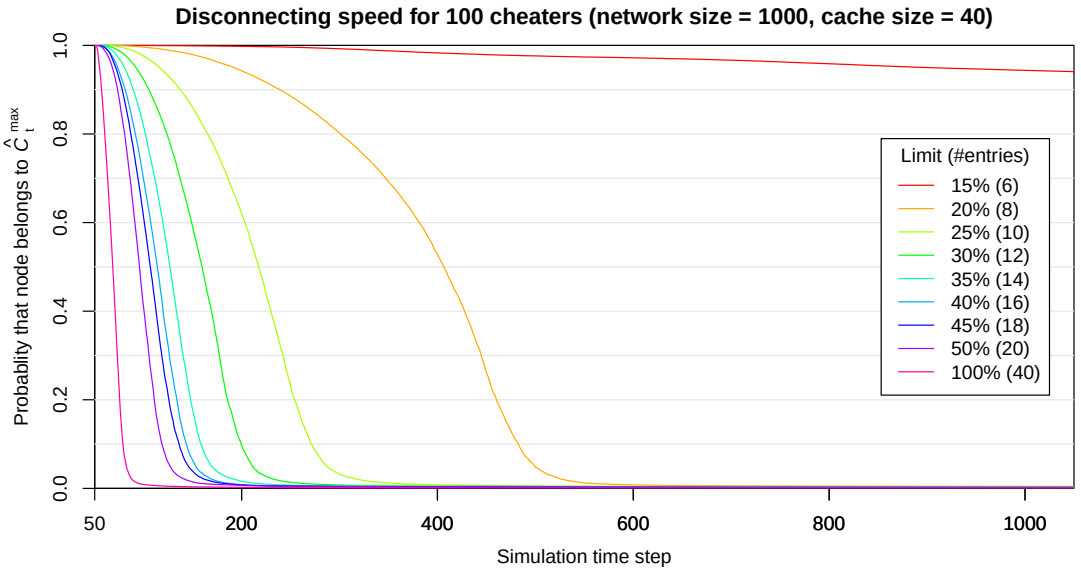


Figure 7.28: Disconnecting speed for 100 cheaters, compared for different limits on the cache exchange for the Newscast network consisting of 1000 nodes with the cache containing 40 entries.

malicious users (or at least with a mechanism to detect their activity). This could be the task assigned for the delegated resources of the DGVCS owner. Then, the scheme could be dynamically adapted to the current situation. For high activity of the cheaters, the limit should be decreased, whereas for the low one — it should return to higher levels. However, application of the approach has its limitation — the number of the exchanged cache entries between the nodes can not be decreased indefinitely as at some point the resulting scheme starts to lose its flexibility.

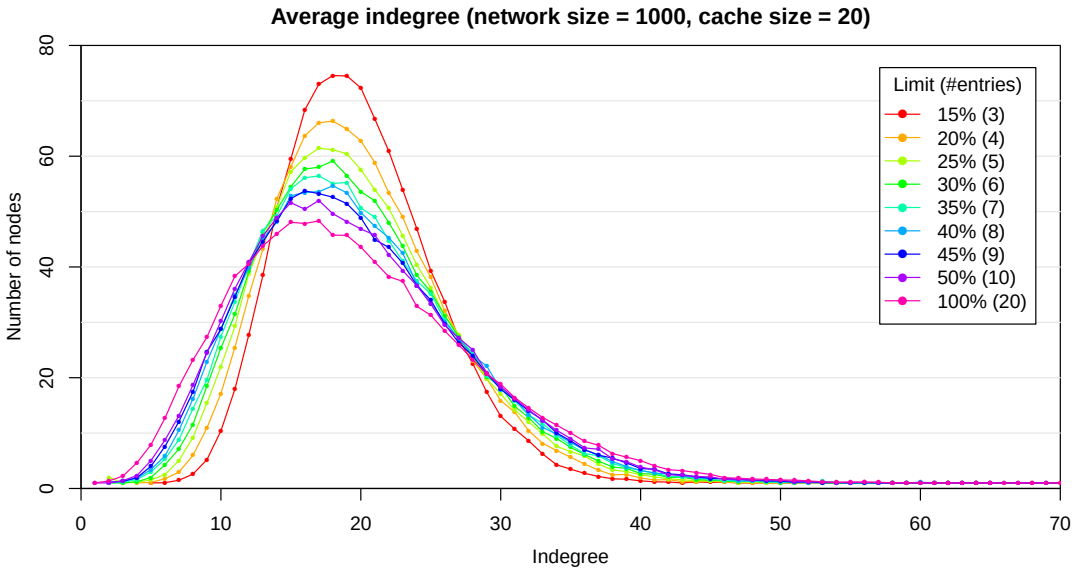


Figure 7.29: Influence of the cache-exchange limit on the average indegree in the Newscast network consisting of 1000 nodes with the cache containing 20 entries.

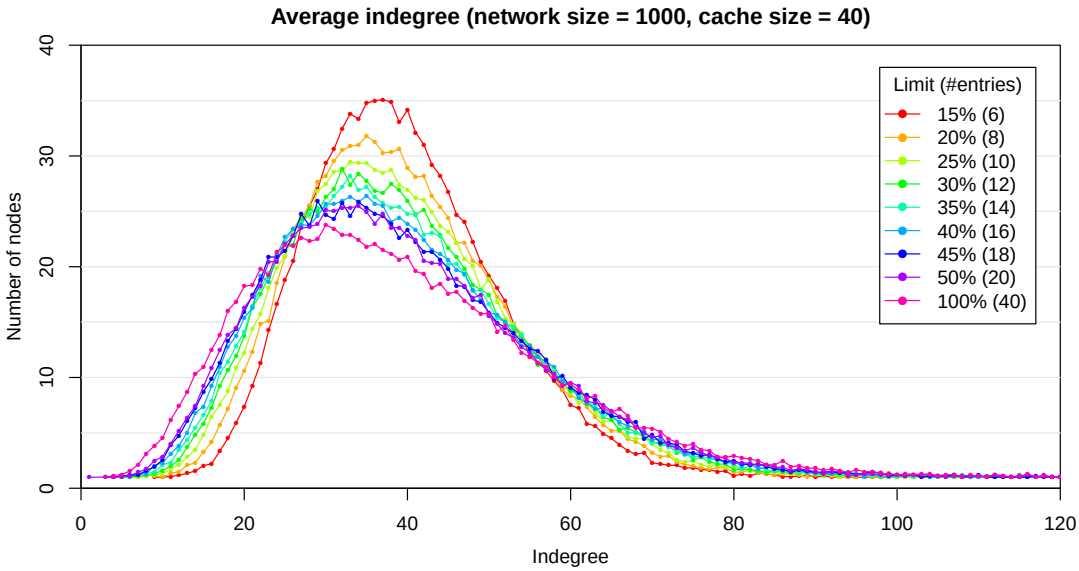


Figure 7.30: Influence of the cache-exchange limit on the average indegree in the Newscast network consisting of 1000 nodes with the cache containing 40 entries.

7.2.4.2 Influence of the limit on the connection graph

Limiting the number of cache entries exchanged during a connection between two nodes does not only affect the cheating clients but also the non-malicious part of the network. Now, we will analyse the impact of this alteration in the protocol on the connection graph.

The first and the most noticeable effect concerns the indegree distribution in the graph. Figures 7.29 and 7.30 present the average value of this statistic through 100 executions at the 50th simulation step. The data was gathered for each combination of the parameters: the network and the cache sizes, and the different limits. The trend is identical for all the

cases, therefore we only present the results for the network consisting of 1000 nodes.
On both figures, the distribution is right-skewed[3] for all the limits. The smaller it the number of the exchanged cache entries, the closer is the peak number of nodes having an indegree close to the cache size. Additionally, the distribution is more narrow with the decreasing limit. Hence, the number of peers in the network having the extreme values of indegree is reduced. These trends are amplified with the decreasing number of the exchanged items.
All that means that the introduced restriction positively affects the load of each node. The expected number of incoming connection is more even. However, this has a side effect manifested by the slower information dissemination.

The small-world properties of the newscast Network are also affected by the introduced limit. Figures 7.31 and 7.32 present the results — the average path length (APL) and the average clustering coefficient (CC). The presented values are the average from 100 executions, measured for networks consisting of 1000 nodes with different limits on the number of exchanged entries. Figure 7.31 presents the data obtained for the cache size set to 20, whereas Figure 7.32 — for 40. As previously, the measurements for smaller networks are characterised by the similar trends, therefore they are omitted from the discussion.
The initial values of the APL are high due to the initialisation phase (described in Section 7.2.1.3). The average path length in a grid consisting of 1000 nodes is approximately 17,88. This value drops to the lower levels with the progress in the execution of the protocol. Hence, the range of the vertical axis was limited to increase the readability of the relevant values.

As visible on Figures 7.31 and 7.32, the time required to reach the self-organised equilibrium in the network extends with the decreasing limit on the number of exchanged entries. Both statistics also converge to lower values. As previously, these trends are amplified with the decreasing number of exchanged entries. Generally, the networks with a bigger cache size are less affected by the change in the protocol mechanics. It is worth noting that the equilibrium is reached in all the cases within the bootstrap time (50 simulation steps) set for the experiments presented before.

7.2.4.3 Summary of the results

Limiting the number of exchanged cache entries during a connection proved to be an effective method against the cheating model introduced in this chapter. Even at 50%, the solution gives satisfactory results. The nodes stay connected in the biggest component much longer without significantly affecting the initial (small-world) properties of Newscast. Yet, further decrease of the limit, allowing tolerating greater number of the malicious clients, has more influence on the protocol operation.

Limiting the cache-exchange to the 30% of the freshest entries proves to tolerate almost completely up to 1% of the malicious users[4] during 1000 simulation steps. On the other hand, decreasing the limit to 15% allows resisting the attack from 100 cheating clients (10%). However, such serious change affects the small-world properties of the connection graph. Therefore, the adjustment should be made dynamically during the execution of the protocol, to minimise the impact of the approach on the executed computation.

[3] Right-skewed distribution — the right tail is longer with the mass concentrated on the left.
[4] 10 cheaters connected to the network consisting of 1000 non-malicious nodes.

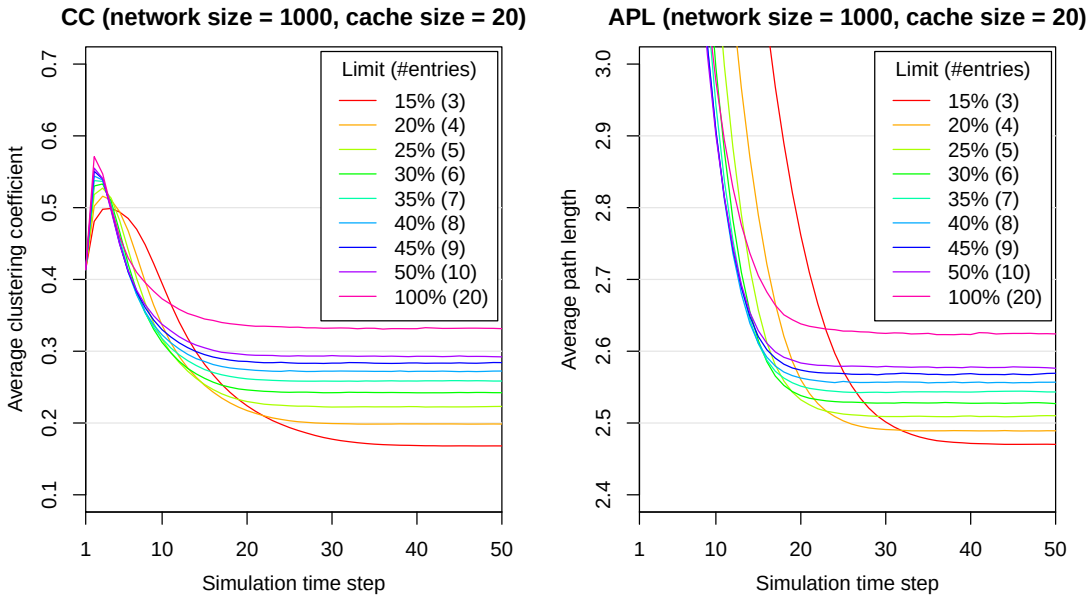


Figure 7.31: Influence of the cache-exchange limit on the average clustering coefficient (CC) and the average path length (APL) in the Newscast network consisting of 1000 nodes with the cache containing 20 entries.

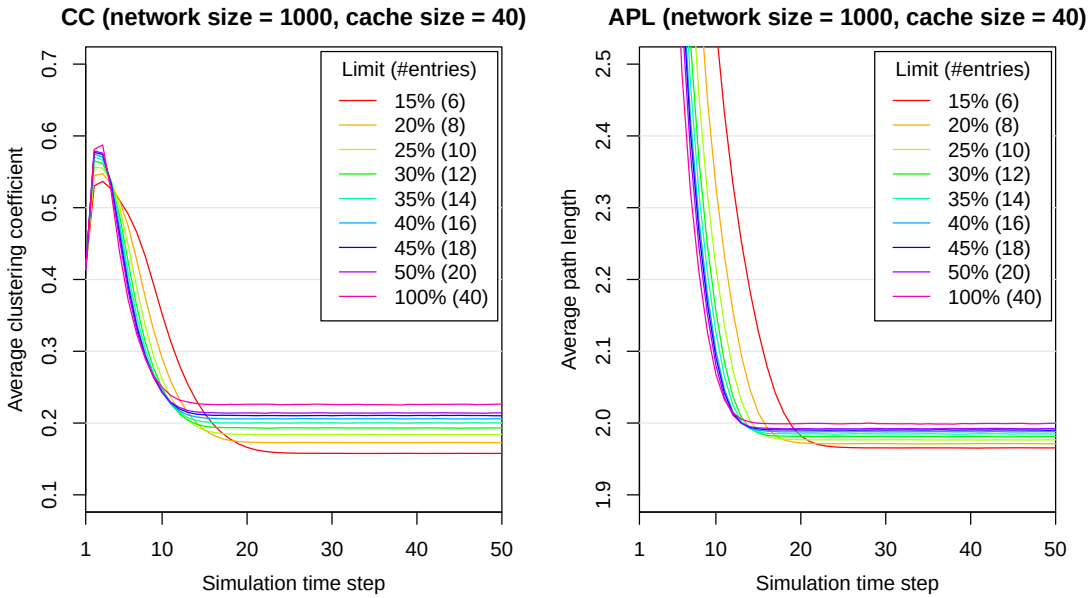


Figure 7.32: Influence of the cache-exchange limit on the average clustering coefficient (CC) and the average path length (APL) in the Newscast network consisting of 1000 nodes with the cache containing 40 entries.

7.3 Summary and perspectives

In this chapter, we have analysed cheating-tolerance of distributed Evolutionary Algorithms (EAs). Our study was mainly focused on Evolvable Agent Model (EvAg) — the only distributed model able to utilise vast computing resources offered by Desktop Grids and Volunteer Computing Systems (DGVCS's). Its reference implementation relies on Newscast — a Peer-to-Peer (P2P) gossip protocol, defining the neighbourhoods at the evolution and the communication layers. Following a probabilistic scheme, the system is able to keep a self-organized equilibrium. The emerging connection graph behaves as a small-world topology allowing a scalable information dissemination, aiding the optimisation search. The existing empirical studies of the whole solution, found in the literature of the subject, indicate its natural resilience to crash failures. This property is inherited from the evolutionary process and the communication protocol.

To the extent of our knowledge, the solution has not been analysed before in the context of cheating faults. As we demonstrated, it is possible to ensure cheating-tolerant execution at the evolution layer but the global success of the optimisation search depends mainly on the communication protocol. Newscast was not designed to be cheating-tolerant and as our experiments demonstrated — it is not. The problem of malicious users is not new in the context of Peer-to-Peer (P2P) networks, nevertheless we did not find a suitable, cheating-tolerant replacement for the protocol.

Therefore, we performed an in-depth analysis of Newscast against cheaters. First, we examined the data flow in the original protocol which revealed and explained the vulnerabilities build-in in the original design. Later, we developed an efficient, non-cooperative malicious client of Newscast — flooding the network with the freshest, random (non-valid) entries. Conducted experiments reveal that the protocol is sensitive to such attack. Its successful execution leads to the graph being split after a relatively short time from the moment when the cheaters start their activities. Finally, we demonstrated that it is possible to decrease the influence of malicious clients by limiting the number of exchanged cache entries to the fraction of the freshest items.

However, the complete solution based on the approach described above requires further research including churn, the asynchronous environment and the cheating-tolerance against cooperative cheaters. The proposed execution scheme aids further extensions with detection and prevention mechanisms, hopefully providing in the future a robust and scalable protocol, creating a solid foundation for the execution of Evolvable Agent Model (EvAg) in Desktop Grids and Volunteer Computing Systems (DGVCS's).

Part III

Conclusion

Chapter 8

Conclusions & Perspectives

Contents

This chapter recalls the context of the dissertation, summarizes the different contributions developed during the thesis and details the challenges and perspectives opened by this work.

8.1 Summary

Evolutionary Algorithms (EAs) have gained popularity for more than 40 years now as a successful framework able to deal efficiently with a wide range of difficult problems. These include classical NP-hard combinatorial and diverse real-world optimisation problems, typically difficult to solve using traditional search methods. Especially the latter can be non-linear, highly constrained, multi-objective and/or include many uncertainties.

Many models of EAs have been proposed over the years — see Chapter 4 for details. However, their theoretical analysis has been long ignored until the seminal work of Günter Rudolph [135] in the 1990s. Since then, major progress have been made, especially in the last 10 years. However none dealing with aspects of fault-tolerance.

In this thesis, we were interested in a better understanding of the theoretical foundations of EA robustness, especially when the execution might be corrupted by malicious acts. Such actions are a common issue in large-scale Desktop Grids and Volunteer Computing Systems (DGVCS's) utilising idle resources shared by volunteers. The incentives offered to the contributors attract also malicious users, commonly called **cheaters**. A cheater typically seeks to obtain the rewards with little or no contribution at all.
Due to the vast computational and storage capabilities provided by DGVCS's at a low cost, many large-scale research projects are carried out using such set-ups. However, this advantage is obtained at the expense of a challenging, error-prone, heterogeneous and volatile environment of execution.

In this context, this work offers a formalisation of **cheating faults** — a subtype of byzantine (arbitrary) faults — modelling the malicious behaviours described above. The actions of cheaters are mainly characterised by the alteration of outputs produced by some or all tasks forming a distributed execution. The approach differs from the arbitrary faults in the

implementation, as cheating faults are introduced intentionally from within the boundaries of a system. More importantly, we propose in this thesis a theoretical analysis of the inherent resilience of EAs against cheaters.
There are two directions in which the theoretical analysis of an EA can be conducted, both focused on some notion of time. One can either study whether the EA is able to find the solution in a finite time (**convergence analysis**) or one might wish to estimate the expected time required by the EA to find the optimal solution of the problem (**expected runtime analysis**). We mainly focused on the first direction in this manuscript. However, throughout the work on this thesis, a complete runtime analysis have been performed for a new type of EA with a population structured in a small-world graph. This led to a significant improvement of the state-of-the-art results proposed in the literature regarding the asymptotic upper bound for the expected parallel runtime.

More generally, this thesis permits revealing the convergence conditions of parallel EAs, a first contribution for which a specific conclusion is proposed in Section 8.1.1. A second contribution lies in the domain of spatially-structured EAs and gossip protocols. The cheating-tolerance analysis was conducted for Evolvable Agent Model (EvAg), relying on Newscast protocol to define neighbourhoods at the evolution and the communication layers. A specific conclusion on this topic is proposed in Section 8.1.2.

8.1.1 Contributions to the cheating-tolerance of parallel EAs

Many previous studies suggested through experiments the innate resilience of EAs against crash faults [37, 64, 75, 111, 166]. In this thesis, the enclosed convergence results by means of Markov chain modelling, offer new theoretical insights on the convergence of EAs despite the presence of cheaters. As mentioned before, **cheating faults** can be modelled as the surreptitious alteration of the output values produced by some or all tasks of the program being executed. Such a selfish behaviour is unfortunately common on Desktop Grid and Volunteer Computing System (DGVCS) platforms and can affect BOINC-based EAs such as the MilkyWay@home project [40, 42, 66].

The provided analysis permits concluding formally on the robustness (or non-robustness) of an EA. Table 6.1 on page 86 summarizes our contributions from this perspective. The fact that there exists some cases where an EA *always* converges despite the presence of cheating faults is quite encouraging. This will promote the usage of EAs in the future developments around distributed computing platforms such as Desktop Grids and Volunteer Computing Systems or Cloud systems where the resources cannot be fully trusted. In particular, our work shows that the modification step of an EA can be "safely" executed on the untrusted workers without any special protection: if cheating is present at this level, it will not affect the convergence of the optimisation search towards valid and (hopefully) optimal solutions. In this sense, EAs can be considered to have an Algorithm-Based Fault Tolerance (ABFT), hence to be fault-tolerant without any extensions. Alternatively, our study also highlights that as soon as cheating happens at the selection level and no special validation mechanisms are used, there is a chance that the algorithm will *not* converge. This means that in this case, additional measures have to be introduced.

8.1.2 Contributions to the cheating-tolerance of distributed EAs and gossip protocols

We have extended the cheating-tolerance analysis to spatially-structured Evolutionary Algorithms (EAs) and gossip protocols. Our study was mainly focused on Evolvable Agent Model (EvAg) — the only distributed model able to utilise vast computing resources offered by Desktop Grids and Volunteer Computing Systems (DGVCS's). Its reference implementation relies on Newscast — a Peer-to-Peer (P2P) gossip protocol, defining the neighbourhoods at the evolution and the communication layers. Following a probabilistic scheme, the system is able to keep a self-organized equilibrium. The emerging connection graph behaves as a small-world topology allowing a scalable information dissemination, aiding the optimisation search. The existing empirical studies of the whole solution, found in the literature of the subject, indicate its natural resilience to crash failures. This property is inherited from the evolutionary process and the communication protocol.

To the extent of our knowledge, the solution has not been analysed before in the context of cheating faults. As we demonstrated, it is possible to ensure cheating-tolerant execution at the evolution layer but the global success of the optimisation search depends mainly on the communication protocol. Newscast was not designed to be cheating-tolerant and as our experiments demonstrated — it is not. Therefore, we performed an in-depth analysis of the protocol against cheaters. First, we examined the data flow in Newscast, which revealed and explained the vulnerabilities build-in in the original design. Later, we developed an efficient, non-cooperative malicious client of the protocol — flooding the network with the freshest, random (non-valid) entries. Conducted experiments reveal that the protocol is sensitive to such attack. Its successful execution leads to the connection graph being split after a relatively short time from the moment when the cheaters start their activities. Finally, we demonstrated that it is possible to significantly decrease the influence of malicious clients. It was achieved by limiting the number of exchanged cache entries to the fraction of the freshest items without substantially affecting the original, small-world properties of the solution.

8.1.3 General conclusion

The theoretical analysis of evolutionary computation has made immense progress during the last 10 years. This Ph.D. comes in this trend, and offer novel perspectives as regards the resilience of EAs against cheating faults. This insight not only leads to a better understanding of the fault-tolerance nature of EAs, but it also provides design guidelines for developing more robust approaches.

By the variety of problems addressed by EAs, this study will hopefully promote their usage in the future developments around distributed computing platforms such as Desktop Grids and Volunteer Computing Systems or Cloud systems where the resources cannot be fully trusted. BOINC-based EA projects such as MilkyWay@home [40, 42] could directly benefit from the outcomes of this work.

Answering the research questions posed in Section 1.2:

- *Can we formally analyse in which conditions an EA is expected to converge (or not) towards valid solutions despite the presence of cheating faults?*
- *Which models for EA executions aid the resilience against cheaters?*
- *And finally, which properties of the models contribute to the innate resilience?*

We have successfully conducted the convergence analysis of parallel EAs subjected to malicious acts. It resulted in specifying the required conditions for the algorithm to guide the optimisation search towards valid and (hopefully) optimal solutions.

Both EA execution models (parallel and distributed) analysed in the context of DGVCS's and cheaters, aid the resilience against malicious users. In either case, robust solutions are possible. In parallel EAs it is easier to verify validity of the results returned by the volunteered resources, however this affects the global progress of the search as possible cheating influences the whole population. On the other hand, in the distributed executions, the population is divided. Therefore, the influence of malicious users is constrained to their neighbourhoods. The responsibility for the global success of the optimisation search is transferred partially on the communication protocol, which has to limit the number of resources affected by cheating.

Finally, the stochastic properties of the algorithm and its execution models contribute the most to the resilience against cheaters. The unpredictable behaviour of the solution makes it difficult to cheat efficiently. Additionally, the division of the population into small subpopulations (or single individuals) greatly limits the possible area of the system affected by the malicious activities.

8.2 Perspectives and future work

In this thesis we proposed a theoretical and practical analysis of the fault-tolerant nature of parallel or distributed Evolutionary Algorithms (EAs), when executed in a distributed environment subjected to malicious acts commonly encountered in Desktop Grids and Volunteer Computing Systems (DGVCS's).

As a future work, our theoretical analysis could be extended to other models of EAs (with different selection and modification schemas). It is also interesting to analyse in which steps of the algorithm cheating has the biggest influence on the evolutionary process and the execution time.

The cheater model could also be further extended to be more sophisticated. Instead of choosing the cheating points at random, they could be selected according to their harmfulness. Cooperation of the malicious users was also not considered in this study and can be of interest for further research.

Moreover, we are now interested in the formal analysis of the overhead induced by the cheating faults. Indeed, the fact that the EA still converges towards valid and optimal solutions despite the malicious tampering with the process, does not mean that this will happen in a reasonable execution time (compared to an execution in a fault-free environment).

Regarding the development around the cheating resilience of spatially-structured EAs and gossip protocols, we plan to build a robust scheme supporting long-lasting executions based on the analysis presented in this thesis. Furthermore, detection techniques of malicious acts could be build-in in the solution to actively remove cheaters from the computation, typically through collaborative blacklisting approaches. The cooperation of the malicious users was also not considered, yet it is a greater threat to a distributed execution than to the parallel one. We plan to develop an efficient, cooperative cheating clients, test our solution against them and propose further improvements. Finally, having the full solution, it is necessary to check how it performs in a more realistic, asynchronous environment, when the dynamics of the peer participation (i.e. churn) is present.

Part IV

Appendix

Appendix A

Table of Notations for Chapter 6

Classical notations found in the literature relative to the convergence of EA are used in this thesis. In particular, most of the conventions taken in this work are directly inspired by the seminal article of Rudolph [136]. They are now briefly reminded.

	Description
$\mathcal{M}$	search space
$f : \mathcal{M} \to \mathbb{R}$	Fitness/objective function minimised by EA, bounded from below $\forall_{x \in \mathcal{M}} f(x) > -\infty$
N	number of individuals
$X_0 = (X_{0,1}, \ldots, X_{0,N})$	initial population (at step $t = 0$) chosen according to some initial distribution $p(.)$; $\forall_{t \geq 0} \forall_{i=1,\ldots,N} X_{t,i} \in \mathcal{M}$
$E = \mathcal{M}^N$	state space
$K(.,.)$	stochastic kernel describing transition of population from step t to $t+1$ for $t > 0$ using so-called genetic operators
$(X_t : t \geq 0)$	Markov chain with values in a set E of a measurable space $(E, \mathcal{A})$
$f^* = min\{f(x) : x \in \mathcal{M}\}$	global optimum

Additional definitions:

- $K^{(t)}(x, A) = \begin{cases} K(x, A) & , t = 1 \\ \int\limits_E K^{(t-1)}(y, A) K(x, \mathrm{d}y) & , t > 1 \end{cases}$ — t-th iteration of the Markovian kernel for any set A;
- $b(X_t) = min\{f(X_{t,i}) : i = 1, \ldots, N\}$ — the best objective/fitness function value of population X_t at step $t \geq 0$;
- f^* — global minimum of the objective/fitness function $f : \mathcal{M} \to \mathbb{R}$;
- $d(X_t) = b(X_t) - f^*$ — distance of the best objective/fitness function value of population X_t at step $t \geq 0$ to the global optimum f^*;
- $A_\epsilon = \{x \in E : d(x) < \epsilon\}$ — set of ϵ-optimal states with $\epsilon > 0$;
- $B(x) = \{y \in E : b(y) \leq b(x)\}$ — set of states which are better than or equal to the state x according to the objective function.

Appendix B

Expected Runtime Analysis in a Fault-free Environment for an elitist parallel Evolutionary Algorithm

One key motivation behind this PhD research is a better understanding of the theoretical foundations of EAs, especially when the execution is corrupted by malicious acts. With the ambition to characterise the impact of cheating faults on the expected runtime of EAs, we were investigating potentially good candidates for a resilient spatially-structured EA. With this in mind, we found a Peer-to-Peer (P2P) EA whose population is structured using Newscast gossip protocol [85].

In the attempt to model and analyse the algorithm, we arrived to a simplified network topology with the small-world properties. This leaded not only to a complete theoretical runtime analysis of this type of EA, it also permitted to improve the state-of-the-art results proposed in the literature. The study [112], were finally the cheating aspects are absent, remains of interest in the scope of this manuscript. Thus, we decided to include it as an appendix. It can also serve as an illustration of a full runtime analysis, complement to the overview provided in the Section 4.4.

Expected Running Time of Parallel Evolutionary Algorithms on Unimodal Pseudo-Boolean Functions over Small-World Networks

Jakub Muszyński, Sébastien Varrette, and Pascal Bouvry

In *Proc. of the IEEE Congress on Evolutionary Computation (CEC 2013)*, pages 2588–2594, Cancún, Mexico, June 2013. IEEE.

Abstract — This paper proposes a theoretical and experimental analysis of the expected running time for an elitist parallel Evolutionary Algorithm (pEA) based on an island model executed over small-world networks. Our study assumes the resolution of optimization problems based on unimodal pseudo-boolean functions. In particular, for such function with d values, we improve the previous asymptotic upper bound for the expected parallel running time from $\mathrm{O}(d\sqrt{n})$ to $\mathrm{O}(d \log n)$. This study is a first step towards the analysis of influence of more complex network topologies (like random graphs created by P2P networks) on the runtime of pEAs. A concrete implementation of the analysed algorithm have been performed on top of the ParadisEO framework and run on the HPC platform of the University of Luxembourg (UL). Our experiments confirm the expected speed-up demonstrated in this article and prove the benefit that pEA can gain from a small-world network topology.

Algorithm B.1: General scheme of an EA in pseudo-code.

```
t ← 0;
Generation(X_t); // generate the initial population
Evaluation(X_t); // evaluate population
while stopping criteria not satisfied do
    X̂_t ← ParentsSelection(X_t); // evaluate population
    X'_t ← Modification(X̂_t); // cross-over + mutation
    Evaluation(X'_t); // evaluate offspring
    X_{t+1} ← Selection(X_t, X'_t); // select survivors for the next generation
    t ← t + 1;
```

B.1 Introduction

Evolutionary Algorithms (EAs) are a class of solving techniques based on the Darwinian theory of evolution [36] which involves the search of a **population** X_t of solutions. Feasible solutions are called **individuals** and members of the population. Each iteration of an EA involves a competitive selection that weeds out poor solutions through the **evaluation** of a fitness value that indicates the quality of the individual as a solution to the problem. The evolutionary process involves at each generation a set of stochastic operators that are applied on the individuals, typically recombination (or cross-over) and mutation. There exists many useful models of EAs, the most popular and widely applied being the sequential EA, for which a pseudo-code of the general execution scheme is provided in Algorithm B.1. In that case, a single population (panmixia) of individuals is used.
Adding parallelism to EAs is relatively simple [159] and implies interesting features as de-

scribed in [4], more precisely:

- the reduction of the time to locate a solution (faster algorithms),
- the reduction of the number of function evaluations (cost of the search),
- the possibility of handling larger populations using parallel platforms for running the algorithms,
- the improved quality of the solutions worked out.

From the large variety of parallel EAs models that exist in the literature [4], this work targets coarse-grain distributed EAs based on an **island model**: several populations connected by a graph structure run in parallel and evolve independently for some time before periodically exchanging part of their best individuals to neighboring island i.e. adjacent population in a migration process. This model is well suited for parallel and distributed execution on computing cluster or grids.

Many experimental results are reported on all types of EAs yet despite the long history and very active research in the area, only few papers in the literature addresses the theoretical foundation of EAs, whether at the level of convergence proofs or running time analysis. In a previous work [114], we studied the inherent resilience of EAs toward crash faults and cheating when executed on a distributed and potentially hostile environment such as a DGVCS platform. In particular, we proved the convergence of the algorithm toward valid solution despite the presence of fault under some conditions. The open question left at that moment was a careful analysis of the expected running time of EAs. When investigating this aspect and searching for potentially good candidate for a resilient distributed EA, we came to a P2P EA whose population is structured using the NewsCast gossiping protocol [82,88,96]. In the attempt to model and analyze the expected running time of this algorithm, we arrived to a simplified network topology close to a small-world graph. This leaded not only to a completed theoretical running time analysis over a new type of EA, it also permits improving the state-of-the-art results proposed in the literature. This article details this analysis which has been performed on the most simple variant of a pEA that is still of theoretical and practical interest: an elitist (1 + 1) pEA where the size of the population is restricted to one individual and where a cross-over step intervene with probability p_c. Its pseudo-code is proposed in the Algorithm B.2.

This paper is organized as follows: section B.2 details the background of this work and defines the small-world network model at the heart of this study. Section B.3 holds the main contribution of this paper as it details the theoretical analysis that exhibit a upper bound on the asymptotic running time of an elitist (1 + 1) pEA with cross-over whose islands are structured over a small-world network. The section B.4 validates the theoretical bound on concrete runs performed on the HPC platform of the University of Luxembourg (UL). Additional experiments are provided that illustrate the speed-up obtained by adding new communication edges to the initial infrastructure. Finally, section B.5 concludes the paper with a summary of our results and future directions.

B.2 Context & Motivation

B.2.1 Optimisation algorithm

We consider the maximization of a unimodal pseudo-boolean function $f : \{0,1\}^n \to \mathbb{R}$ (see Definitions B.2.1 and B.2.2), where n denotes the number of bits. The optimization algorithm is pEA (see Algorithm B.2). We assume that there is a topology given by an undirected

Algorithm B.2: Elitist parallel (1 + 1) EA with crossover.

```
// initialization
for i ← 1 to number of islands (μ) do
    x^i ← uniform random sequence {0,1}^n;
// evolution
while optimal solution is not found do
    parallelfor i ← 1 to number of islands (μ) do
        y^i ← x^i with each bit flipped with probability 1/n;
        if f(y^i) ≥ f(x^i) then x^i ← y^i;
        broadcast x^i to all neighbours;
        z^i ← migrant with maximum fitness value;
        z^i ← crossover of z^i and x^i with probability p_c;
        if f(z^i) ≥ f(x^i) then x^i ← z^i;
```

graph, where vertices are islands and edges indicate neighbourhoods between them. Each island (i.e. vertex, node) is responsible for evolutionary process of one population (in our case one individual) and communication with its neighbours in the topology.

Definition B.2.1 (Pseudo-boolean function [178])**.** *A pseudo-boolean function* $f : \{0,1\}^n \to \mathbb{R}$ *is a degree-k function with* N *non-vanishing terms if it can be represented as*

$$f(x_1, \ldots, x_n) = \sum_{1 \le i \le N} w_i \prod_{j \in S_i} x_j \tag{B.1}$$

where $w_i \in \mathbb{R} - \{0\}$ *and the size of the sets* $S_i \subseteq \{1, \ldots, n\}$ *is bounded above by* k*. Degree-1 functions are called linear and degree-2 functions are called quadratic.*

Definition B.2.2 (Unimodal function)**.** *Unimodal functions are those functions where the global optimum is unique and can be reached from each point from the search space by 1-bit mutations.*

For the experimental part we consider in this paper function LO (leading ones) — measuring the length of the longest prefix consisting of ones only (see Eq. B.2) — as a concrete example for our theorems.

$$\mathrm{LO}(x_1, \ldots, x_n) = \sum_{i=1}^{n} \prod_{j=1}^{i} x_i \tag{B.2}$$

B.2.2 Model of a Small-World Network

The original Small-world network model proposed by Watts and Strogatz [176] consists of a regular lattice (e.g. ring lattice) with some edges rewired at random with probability p (see Fig. B.1). This approach allows to "tune" the graph between regularity ($p = 0$) and disorder ($p = 1$). Such construct exhibits a high clustering coefficient (found in social networks) and a low average path length (like in true random graphs), even for quite small values of p. In context of information spreading, rewiring creates a kind of short-cuts between different parts of the graph.

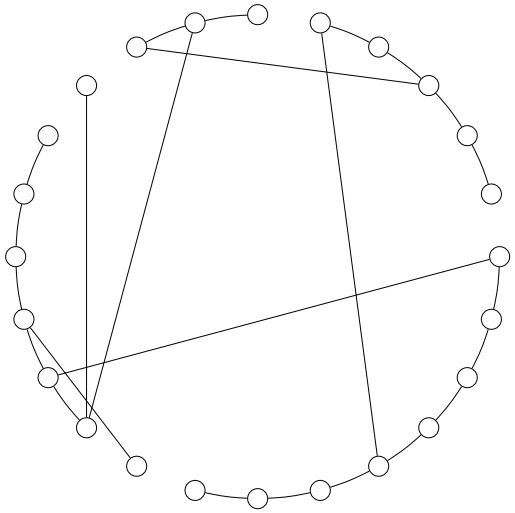


Figure B.1: An example of small world-graph with edges rewiring consisting of 24 nodes. Parameters: $d = 1$, $k = 1$, $p = 0.25$.

Model described above causes problems with an analytic treatment as emphasized by Newman and Watts in [122]. First of all, distribution of short-cuts is not completely uniform (e.g. multiple edges between two vertices are prohibited), therefore average over different realizations of the randomness is hard to perform. Secondly, there is a finite probability that the graph will be split during the process of rewiring the edges. Therefore, average distance between pairs of vertices of the graph and number of quantities and expressions are poorly defined.

To circumvent these problems, a new small-world model is proposed [122]. Starting from regular lattice, instead of rewiring each edge with probability p, short-cuts are added between pairs of vertices chosen uniformly at random. There is also no prohibition on the existence of loops (edge from the given vertex to itself) and multiple edges between two vertices. To preserve compatibility with the results of Watts and Strogatz, short-cuts are added with probability p for each existing edge on the original lattice (see Fig. B.2). This model is equivalent to the Watts-Strogatz model for small p, whilst being better behaved when p becomes comparable to 1.

B.3 Upper bound on the expected running time of an elitist (1 + 1) EA with cross-over built on top of a Small-World network

Our theoretical analysis is based on the approach proposed by Lässig and Sudholt [98] derived from fitness levels method [178]. In this technique, search space is divided into sets $A_1, \ldots, A_m$ called *fitness levels* that are ordered w.r.t. fitness values (i.e. $A_1 <_f A_2 <_f \cdots <_f A_m$, see Definition B.3.1). We say that an island is in A_i or on level i if the current individual is in A_i. In elitist EA (our case, see Algorithm B.2), fitness value in the whole population can never decrease. Therefore, if one can derive lower bounds on the probability of leaving a specific fitness level towards higher levels, this yields an upper bound on the expected running time (see Theorem B.3.1).

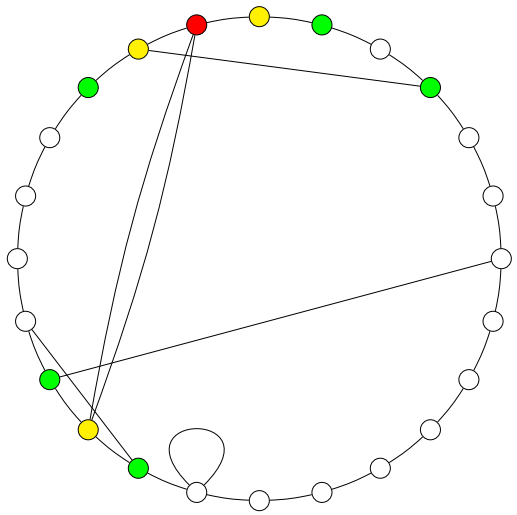


Figure B.2: An example of the small-world graph with short-cuts adding, consisting of 24 nodes. Parameters: $d = 1$, $k = 1$, $p = 0.25$. For the current (red ●) node, all the green ● nodes belong to its neighbourhood surface of radius 2 ($A(2)$). Neighbourhood volume of radius 2 ($V(2)$) for this node, consists of red ●, yellow ● and green ● nodes.

Definition B.3.1 ($<_f$-partition)**.** *For $A, B \subseteq \{0,1\}^n$ and $f : \{0,1\}^n \to \mathbb{R}$ the relation $A <_f B$ holds if $f(a) < f(b)$ for all $a \in A$ and $b \in B$. An $<_f$-partition of $\{0,1\}^n$ into non-empty sets $A_1, \ldots, A_m$ is when $A_1 <_f A_2 <_f \cdots <_f A_m$ and all $a \in A_m$ are global optima.*

Theorem B.3.1 (Fitness-level method for pEAs [98])**.** *Consider a partition of the search space into fitness levels $A_1 <_f A_2 <_f \cdots <_f A_m$ such that A_m only contains global optima. Let s_i be (a lower bound on) the probability that a fixed island running an elitist EA creates a new offspring in $A_{i+1} \cup \cdots \cup A_m$, provided the island contains a search point in A_i. Let μ_t for $t \in \mathbb{N}$ denote (a lower bound on) the number of islands that have discovered an individual in $A_i \cup \cdots \cup A_m$ in the t-th generation after the first island has found such an individual (i.e. a lower bound on the number of **informed islands**). Then the expected parallel running time of the pEA on f is bounded by*

$$\mathrm{E}[T^{par}] \leq \sum_{i=1}^{m-1} \sum_{t=0}^{\infty} (1 - s_i)^{\sum_{j=1}^{t} \mu_t} \tag{B.3}$$

Proof. See [98]. □

For the current best fitness level i, let $\psi_i(x)$ denote the random number of generations until at least x islands are informed. Then expected parallel running time of the pEA can

be further bounded by

$$\mathrm{E}[T^{\mathrm{par}}] \leq \sum_{i=1}^{m-1} \left(\psi_i(x) + \sum_{t=\psi_i(x)}^{\infty} (1-s_i)^{\sum_{j=1+\psi_i(x)}^{t} x} \right) \tag{B.4}$$

$$\leq \sum_{i=1}^{m-1} \left(\psi_i(x) + \sum_{t=0}^{\infty} (1-s_i)^{\sum_{j=1}^{t} x} \right) \tag{B.5}$$

$$= \sum_{i=1}^{m-1} \psi_i(x) + \sum_{i=1}^{m-1} \sum_{t=0}^{\infty} (1-s_i)^{\sum_{j=1}^{t} x} \tag{B.6}$$

i.e. it is a time required to spread the information about individual at fitness level i to x islands plus the remaining optimization time from this point.

So now we need to find an estimation of the time required to spread the information $\psi_i(x)$ in the modified small-world network model proposed in [122]. Assuming that whole communication happens in parallel, $\psi_i(x)$ can be bounded by the radius r of a neighbourhood of volume $V(r)$ (defined on the graph), consisting of at least x vertices ($x \leq V(r)$), multiplied by the expected time required to inform a single island. Lets define p_+ [98] as the lower bound on the probability of informing a vertex in the network. Then the expected time to inform a single island is bounded from above by $1/p_+$. Therefore

$$\psi_i(x) \leq \frac{r}{p_+} \tag{B.7}$$

Following the analysis presented in [122], we have:

$$V(r) = \sum_{r'=0}^{r} a(r') \left[1 + 2\xi^{-d} V(r-r') \right] \tag{B.8}$$

Where:

- $a(r)$ — the surface area of a "sphere" of radius r on the underlying lattice of the model, i.e. the number of nodes which are exactly r steps away from any vertex.
- ξ — length-scale; the typical distance between the ends of short-cuts on the lattice.

$$\xi = \frac{1}{(\phi k d)^{\frac{1}{d}}} \tag{B.9}$$

- k — number of next-nearest neighbours on the underlying lattice (for the full explanation see [122, section II]).
- d — dimension of the underlying lattice.

For one dimension ($d = 1$, a ring lattice) and $k = 1$, we have:

- $a(r) = 2$ for all r.
- $\xi = \frac{1}{\phi}$.

Approximating the sum with an integral and then differentiating with respect to r, we get

$$\frac{\mathrm{d}V}{\mathrm{d}r} = 2 + 4\phi V(r) \tag{B.10}$$

which has a solution (with boundary condition $V(0) = 0$)

$$V(r) = \frac{\mathrm{e}^{4\phi r} - 1}{2\phi} \tag{B.11}$$

Assuming that $0 < \phi \leq 1$ and rearranging for r, we have the value which we searched for

$$r = \frac{\ln\left(2\phi V(r) + 1\right)}{4\phi} \tag{B.12}$$

Therefore

$$\psi_i(x) \leq \frac{\ln\left(2\phi x + 1\right)}{4\phi p_+} \tag{B.13}$$

Theorem B.3.2. *For every unimodal function with d function values*

$$\mathrm{E}[T^{par}] = \mathrm{O}\left(d \log n\right)$$

when executed using the elitist parallel (1 + 1) EA with crossover (where $p_c \leq 1 - \Omega(1)$), on a modified small-world network based on ring lattice with $k = 1$, if $\mu \geq n$.

Proof. An informed island informs uninformed neighbour when no crossover is performed, hence $p_+ \geq 1 - p_c$. We choose an $<_f$-partition of the search space into subsets $A_1 <_f A_2 <_f \cdots <_f A_d$, where A_i contains all search points with the i-th smallest function value. The probability of improving the fitness from level i is at least

$$s_i \geq \frac{1}{n} \cdot \left(1 - \frac{1}{n}\right)^{n-1} \geq \frac{1}{\mathrm{e}n} \tag{B.14}$$

since there is at least one search point in the next fitness level which is at Hamming distance one.

Setting $x = n$ and substituting all of the variables with their bounds in Eq. B.6, gives:

$$\mathrm{E}[T^{\mathrm{par}}] \leq \sum_{i=1}^{d-1} \frac{\ln\left(2\phi n + 1\right)}{4\phi(1 - p_c)} + \sum_{i=1}^{d-1} \sum_{t=0}^{\infty} \left(1 - \frac{1}{\mathrm{e}n}\right)^{tn} \tag{B.15}$$

Since

$$\frac{\ln\left(2\phi n + 1\right)}{4\phi(1 - p_c)} = \mathrm{O}\left(\log n\right) \tag{B.16}$$

and

$$\sum_{t=0}^{\infty} \left(1 - \frac{1}{\mathrm{e}n}\right)^{tn} = \frac{1}{1 - \left(1 - \frac{1}{\mathrm{e}n}\right)^{n}} \tag{B.17}$$

which has a limit with n going to infinity

$$\lim_{n \to \infty} \frac{1}{1 - \left(1 - \frac{1}{\mathrm{e}n}\right)^{n}} = 1 + \frac{1}{\mathrm{e}^{\frac{1}{\mathrm{e}}} - 1} \tag{B.18}$$

We finally get

$$\mathrm{E}[T^{\mathrm{par}}] = \mathrm{O}\left(d \log n\right) \tag{B.19}$$

□

Algorithm	Lattice	Bound	Condition
State of the art results:			
(1 + 1) EA	—	$\mathrm{O}(dn)$	—
parallel (1 + 1) EA	ring	$\mathrm{O}\left(dn^{\frac{1}{2}}\right)$	$\mu \geq (en)^{\frac{1}{2}}$
parallel (1 + 1) EA	grid	$\mathrm{O}\left(dn^{\frac{1}{3}}\right)$	$\mu \geq (en)^{\frac{2}{3}}$
parallel (1 + 1) EA	full graph (K_μ)	$\mathrm{O}\left(d + \frac{dn}{\mu}\right) = \mathrm{O}(d)$	$\mu \geq \Omega(n)$
Contribution of this article:			
parallel (1 + 1) EA	ring + short-cuts	$\mathrm{O}(d \log n)$	$\mu \geq n$

Table B.1: Comparison with existing results for unimodal pseudo-boolean functions.

Parameter		Value
Number of executions		1000
(p) probability of a short-cut	minimum	0.0
	maximum	1.0
	step size	0.1
(μ) population size	minimum	10
	maximum	200
	step size	10
(n) problem size	minimum	10
	maximum	200
	step size	10
(p_c) probability of one-point crossover		0.3
Probability of gene mutation		$1/n$

Table B.2: Parameters used to execute the experiments.

Table B.1 presents comparison with the existing results (see [98]). As it is visible, our proposition improves the upper bound compared to the state-of-the-art solutions at relatively low cost of additional communication over the short-cuts. **Presented approach can be used to derive upper bounds to other lattices (different values for parameters d and k).**

B.4 Experimental Validation

Proposed solution was validated experimentally to see the impact of the number of short-cuts (probability p) on the running time of pEAs. For the purpose of the tests, the algorithm (see Algorithm B.2) was implemented using ParadisEO[1] library. We maintained compatibility with the theoretical model of the network in the experiments (see section B.2.2, description of the small-world graph with short-cuts adding). All the tests were executed on the HPC@Uni.lu[2] platform. Parameters for the executions are gathered in Table B.2. Results are presented on Fig. B.3–B.8. It is worth to note here, that for:

[1] `http://paradiseo.gforge.inria.fr`
[2] `https://hpc.uni.lu`

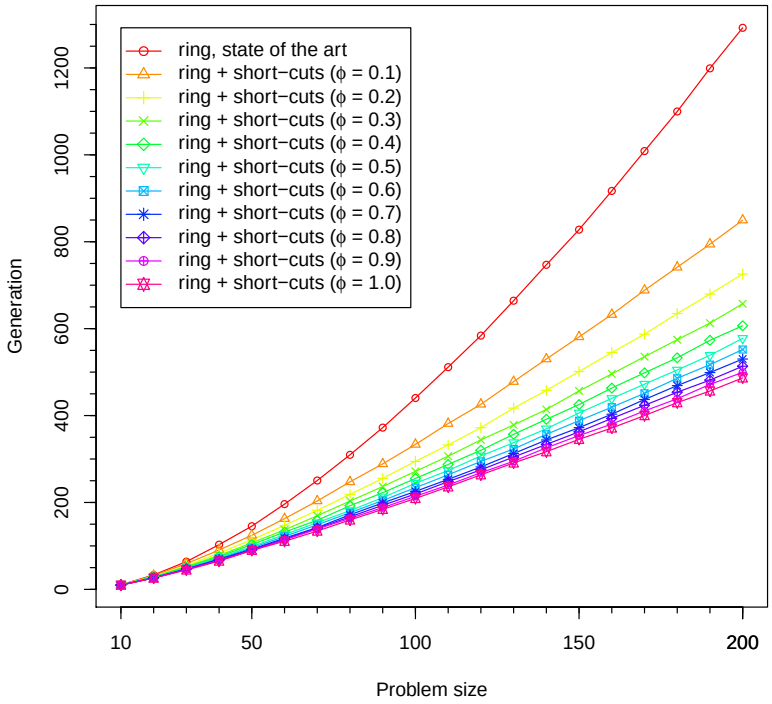


Figure B.3: Results for the function LO (Eq. B.2). The population size (μ) is equal to the problem size (n) in every test case.

- Fig. B.3 and Fig. B.6 the population size (μ) is equal to the problem size (n) in every test, which is exactly the case in the theoretical model,
- Fig. B.4 and Fig. B.7 the population size (μ) is fixed and equal to 200, which is compatible with the theoretical model,
- Fig. B.5 and Fig. B.8, population size (μ) is equal to 100 through the tests, which is not compatible with the theoretical model.

As it is clearly visible on all of the figures mentioned above, even for small values of p we get significant improvement on the convergence time in all of the cases. For example on Fig. B.3, with the problem size equal to 200, from around 1300 generations we go down to around 850 generations (on average over 1000 executions) just by adding 10% of new edges to the original lattice ($p = 0.1$, which creates 20 additional connections in the network). This gives speed-up of around 35%, which is visible on Fig. B.6. A described tendency is maintained for all of the tests.

As Fig. B.5 indicates, there is still some place for improvement in the condition on the minimum population size (μ). For the problem sizes (n) greater than 100, population size (μ) is too small according to the theory, but still the tendency described above is maintained, even though execution time is longer (measured in number of generations). It is also visible on all of the figures, that for the probabilities of short-cuts (p) greater than 0.3, improvement starts to be lower.

Experimental results could be further enhanced by restricting the types of permitted additional connections — loops and multiple edges can be easily avoided.

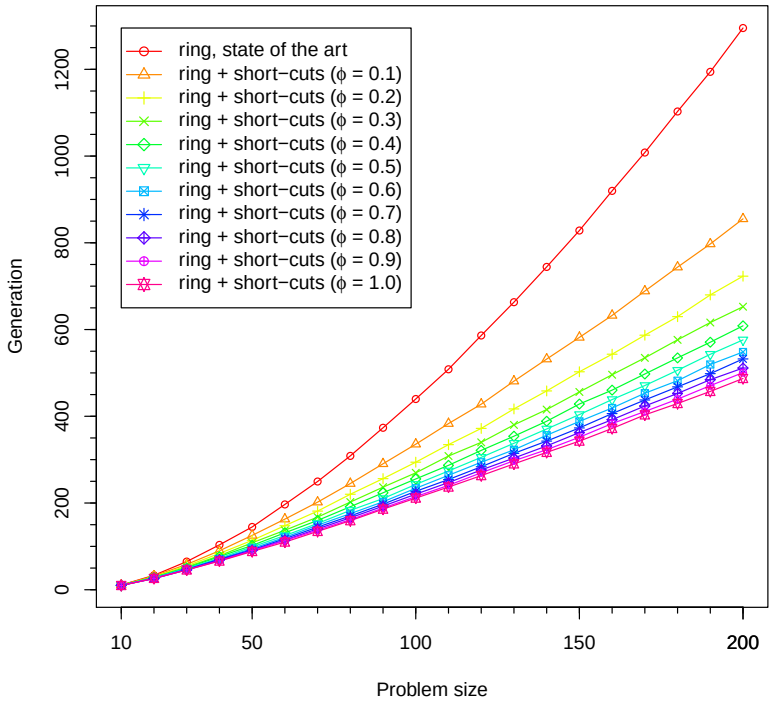


Figure B.4: Results for the function LO (Eq. B.2). The population size (μ) is fixed and equal to 200 in every test case.

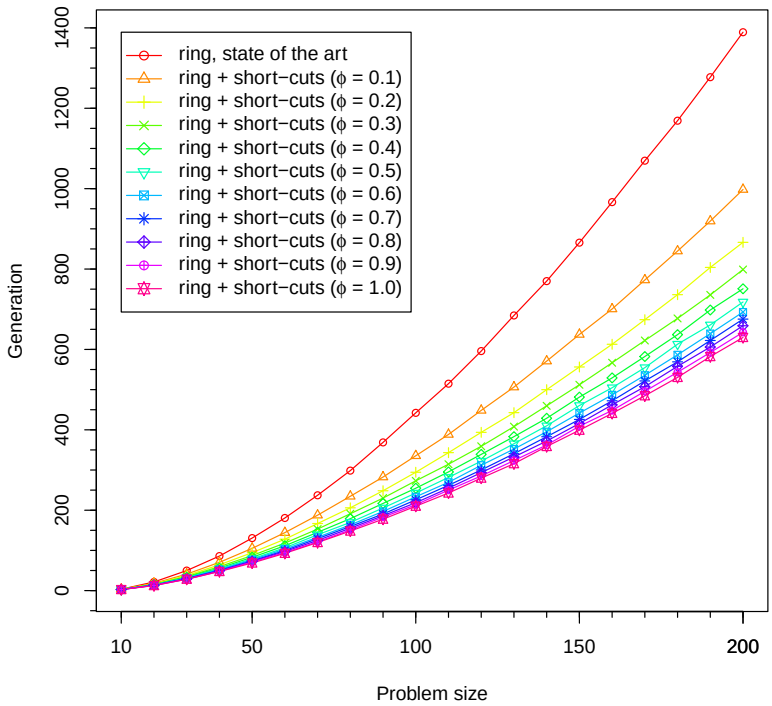


Figure B.5: Results for the function LO (Eq. B.2). The population size (μ) is fixed and equal to 100 in every test case.

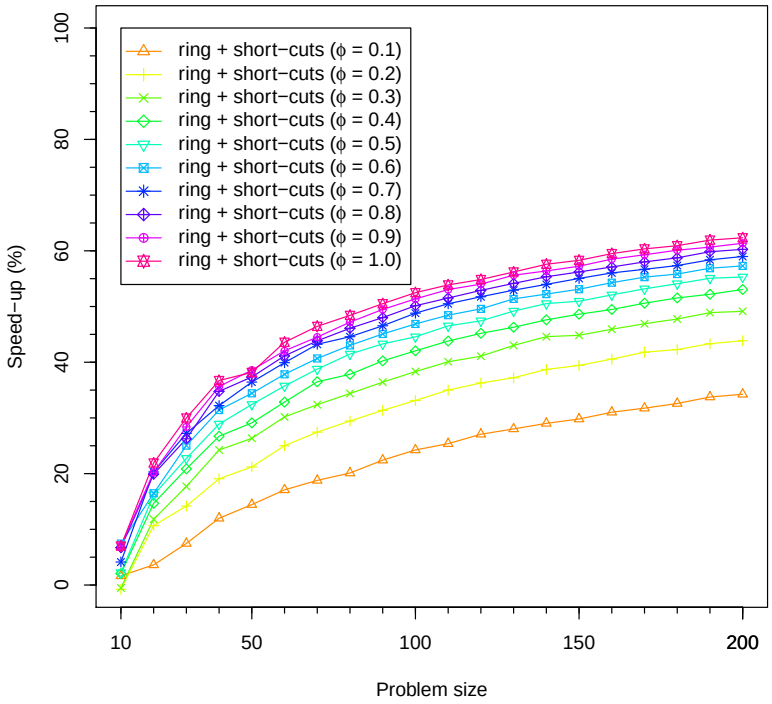


Figure B.6: Speed-up for the function LO (Eq. B.2) compared to a basic ring lattice. The population size (μ) is equal to the problem size (n) in every test case.

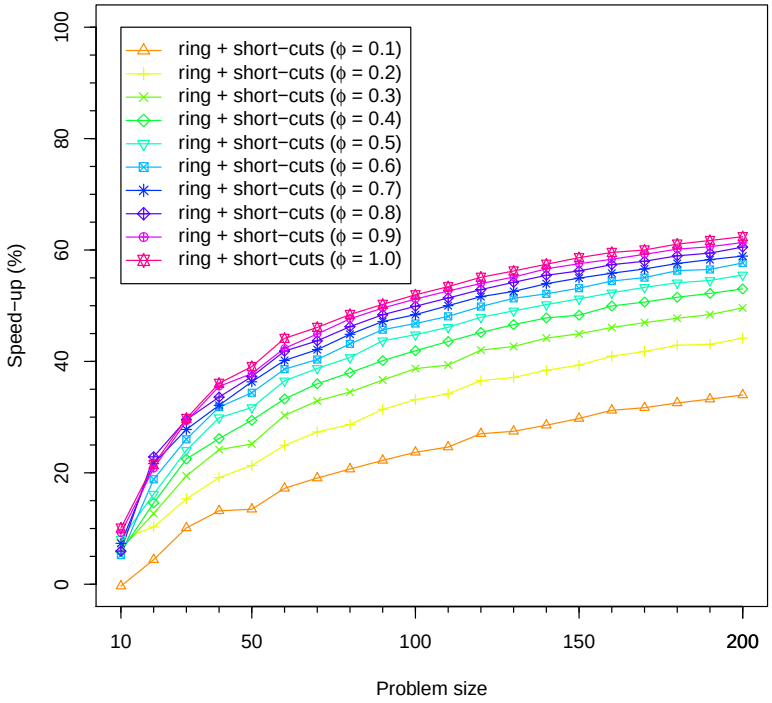


Figure B.7: Speed-up for the function LO (Eq. B.2) compared to a basic ring lattice. The population size (μ) is fixed and equal to 200 in every test case.

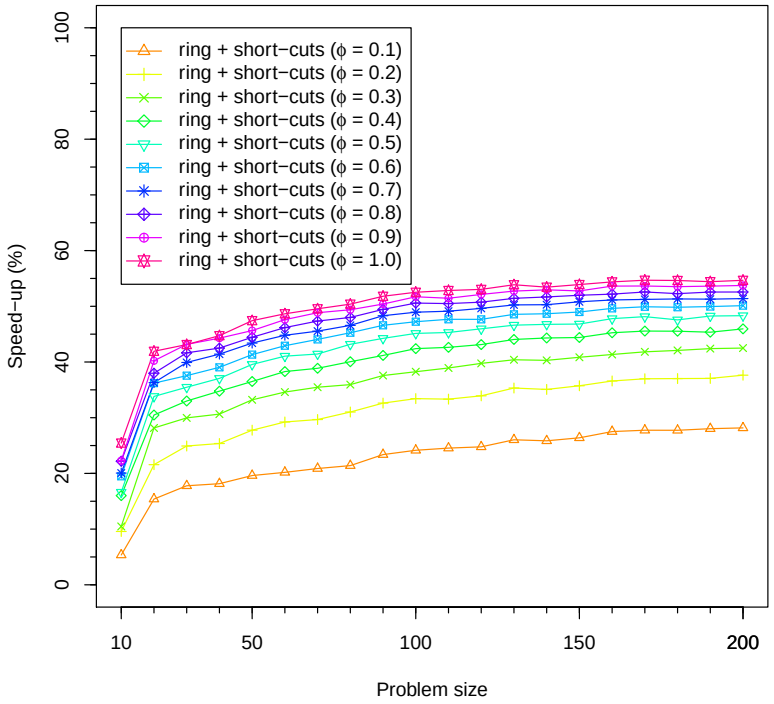


Figure B.8: Speed-up for the function LO (Eq. B.2) compared to a basic ring lattice. The population size (μ) is fixed and equal to 100 in every test case.

B.5 Conclusion & future work

In this paper convergence of pEAs executed over small-world networks was analysed. Proposed solution allows to increase convergence speed of pEAs with relatively low increase in communication cost. Only few additional connections in the network decrease convergence time significantly.

Theoretical analysis included in this paper provides upper bound on pEAs running time of order $\mathrm{O}(d \log n)$ for unimodal pseudo-boolean functions with d function values. Conducted experiments confirm expected speed-up, which in certain cases can exceed 50%.

In future work we plan to extend presented study for more complex connection graphs, like P2P networks generated by Newscast protocol used in highly scalable Evolutionary Computations [88,96,97]. Finally finishing with resilience analysis of pEAs executed in P2P environment, where like in every volunteer computing network, malicious/erroneous acts are common [90].

Acknowledgment

The authors would like to thank Juan Luis Jiménez Laredo for his constructive comments on this work.

Appendix C

A Framework for Description of Gossip Protocols

Contents

Gossip protocols define a pure P2P network (see Section 3.2.2) over a large set of computing resources. The essence of the solution lays in the exploitation of randomness to virally disseminate information and to maintain connectivity in a self-organised (independent of the initial state) equilibrium. Such equilibrium emerges from the loosely-coupled and distributed run of the protocol within different and independent nodes. The epidemic nature provides high fault resilience and self-healing properties at the cost of an overhead in terms of messages routing performance [83].

We start this section with the introduction of a (slightly modified) generic gossip protocol proposed by Jelasity et al. in [83]. We show and describe how the partial knowledge of the global membership (i.e. the partial view, see Section 3.2.2) is build, maintained and exchanged by each peer. Despite the simplicity of the framework, it can be used to describe many of the existing epidemic protocols, facilitating their analysis and comparison.

After the introduction of the general model, we move to a discussion about the metrics and fault tolerance of the emerging network. There we describe how different settings for local management and exchange of gossips influence these properties.

Cheating faults and other malicious activities are a well-known problem in the area of P2P networks and in particular, their epidemic models. At the end of this section, we discuss different attack models leading to the performance degradation, disruptions in communication, data loss, etc.

C.1 Generic gossip protocol

P2P protocols are oriented around the maintenance and management of the partial views of the peers. As defined in Section 3.2.2, a partial view is a list-like structure which contains the addresses or identifiers of some other members of the network. It represents the local (partial) knowledge of a peer about the global membership. In the context of gossip protocols, the partial view is called a **cache** and its content — the **cache entries** — are extended with additional information (see Figure C.1):

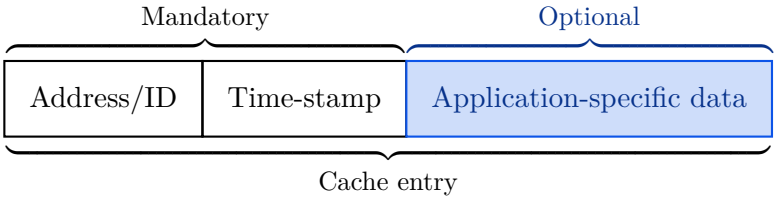


Figure C.1: Cache entry in a gossip protocol.

Algorithm C.1: Definition of the select function for the cache of a generic gossip protocol.

```
method cache.select(c, H, S, list):
    cache.append(list);
    cache.removeDuplicates();
    cache.removeOldItems(min(H, cache.size() − c));
    cache.removeFromHead(min(S, cache.size() − c));
    cache.removeAtRandom(cache.size() − c);
```

- a mandatory **time-stamp** holding the creation time of an entry (sometimes the age of an entry is used instead),
- an optional **application-specific data** which is disseminated virally within the network.

The cache provides standard functionality of a list data structure extended with additional operations specific for the protocol:

- `size()` — returns the **current size** of the cache,
- `append(`*list*`)` — appends a *list* of cache entries and the end of the cache,
- `permute()` — randomly changes the order of cache entries within the current content of the cache,
- `selectPeer()` — randomly chooses a peer to initiate the cache exchange,
- `removeDuplicates()` — removes duplicated entries (the freshest are kept),
- `removeOldItems(`n`)` — removes $n \in \mathbb{N}$ oldest cache entries,
- `removeFromHead(`n`)` — removes $n \in \mathbb{N}$ entries from the head of the cache (not necessarily the youngest/freshest)
- `removeAtRandom(`n`)` — removes $n \in \mathbb{N}$ random cache entries,
- `moveOldestToTail(`n`)` — move $n \in \mathbb{N}$ oldest cache entries to the end of the cache,
- `head(`n`)` — create a sub-list consisting of n first cache entries,
- `select(`c, H, S, *list*`)` — updates the content of the cache with the entries from the *list*, following the constraint on the cache size $c \in \mathbb{N}$, utilising additional parameters $H \in \mathbb{N}$ and $S \in \mathbb{N}$ defining specific behaviour of the protocol (introduced later in detail). The definition of this method is presented in Algorithm C.1.

Two **threads** are used for the protocol's execution: **active** (Algorithm C.2) and **passive** (Algorithm C.3), where the first one is responsible for initiating the communication (i.e. **cache exchange**), the second — for waiting and responding to the requests. Properties of the specific protocol's implementation depends on the following parameters: *push*, *pull*, $c \in \mathbb{N}$, $H \in \mathbb{N}$, $S \in \mathbb{N}$, and $T \in \mathbb{N}$.

The *push* and *pull* define the direction of the communication. Jelasity et al. in [83] demon-

Algorithm C.2: Active thread of a generic gossip protocol.

```
while true do
    wait for T time units; // T — synchronisation time
    p ← cache.selectPeer();
    buffer ← ∅; // empty list
    if push then
        cache.permute();
        cache.moveOldestToTail(H);
        ownEntry ← (ownAddress, currentTime, data); // create a new cache entry
        buffer.append(ownEntry);
        buffer.append(cache.head(c/2 − 1));
    send buffer to p;
    if pull then
        receive buffer_p from p;
        cache.select(c, H, S, buffer_p);
```

strated that the *push-pull* model should be used, as the *push*-only and *pull*-only schemes can partition the network. Additionally, the *pull*-only approach is not very useful in practice, because it is impossible for the newly joined peers to spread any messages or information about themselves to the previously connected nodes.

The size of each cache is limited by the parameter c. However, this constraint is fulfilled only at the ends of the while loops in Algorithms C.2 and C.3. During their execution, the cache may grow beyond c cache entries. Additionally, it is guaranteed that any subsequent removals of entries in the `select` method (Algorithm C.1) will not decrease the size of the partial view below c entries.

Alongside with c, parameters S and H control the final characteristics of the implemented protocol. "Parameter S controls the priority that is given to the addresses received from the peer" [83], i.e. "the diversity of the union of the two new views" [83]. Its name comes from **swap**, as it is the actual number of the exchanged cache entries. H on the other hand, comes from **healing**, as it influences the **self-healing** property of the protocol. It controls "the number of the oldest cache entries moved to the end of the cache" [83] and the number of those removed during the *pull* phase of the execution. Its value should be below or equal to $c/2$ defined together with S from 0 to $c/2 - H$.

Frequency of communication's initiations is configured by the parameter T. If the implementation of the protocol is synchronous, then it is the time after which each peer initiates a single, outgoing connection.

Although the framework defined above is quite general and multiple gossip-based protocols fit into its frame, some minor changes are needed in specific applications. For example, if the peer selection schema is different from random, then the method `selectPeer()` has to be redefined. Or for instance, if some functionality is not needed, then it may be removed (like `permute()`, if the order of cache entries does not have to be altered before sending the view to the other peer).

Finally, the last key functionality to be defined is the **membership management**. In the simplest schema, a node has to connect to any peer from the network to join the system and to leave — cease the communication. In more complex solutions, designated servers

Algorithm C.3: Passive thread of a generic gossip protocol.

```
while true do
    receive buffer_p from p;
    if pull then
        cache.permute();
        cache.moveOldestToTail(H);
        ownEntry ← (ownAddress, currentTime, data); // create a new cache entry
        buffer.append(ownEntry);
        buffer.append(cache.head(c/2 − 1));
        send buffer to p;
    cache.select(c, H, S, buffer_p);
```

may participate in the protocol's execution, gathering the global view (see Section 3.2.2 for the exact definition) of the network. Each joining peer may contact any of such nodes to get from it the initial content of the cache.

C.2 Properties of the protocol induced by the specific settings

In [83], Jelasity et al. analysed different settings of the parameters described in the previous part and their influence on the emerging network properties. The study includes mainly the connectivity, the in-degree distribution, the average path length, and the clustering coefficient (see Section 2.2 for detailed definitions). Additionally, a behaviour of protocols in various configurations is analysed in presence of churn (see Section 5.1.1) and crash failures (see Section 3.3.1).

Next to the crucial *push-pull* policy described earlier, a very important setting is the maximum cache size c. Unfortunately, there is no clear guideline for its choice, because it heavily depends on the specific protocol's instance. For example, in a simple Newscast [82] model, the minimum suggested value of c is 20, which guarantees with a high probability that the emerging network will not partition over time independently of the number of peers (the result was obtained experimentally in [82]).

Due to the fact that the parameters H and S are linked, and in addition must take integer values, the number of possible states is limited (see Figure C.2). The properties of protocols defined at the extreme points of the triangle are as follows [83]:

- **Blind** ($H = 0, S = 0$) — each cache contains a random, never changing subset of peers. This setting does not produce a useful protocol, because it is not able to adopt to changes of the peers' availability.
- **Healer** ($H = c/2, S = 0$) — each cache holds the freshest possible entries.
- **Swapper** ($H = 0, S = c/2$) — cache contents are exchanged between the nodes (up to $c/2 - 1$ entries).

If it comes to the protocols between these extremes, the following properties apply [83]:

- The higher the value of H, the fewer dead links are present in the cache and the faster they are eliminated from the network (in case of crash failures and churn). Therefore, the H should be set as high as possible. Additionally, in realistic churn scenarios, the protocol performs similarly to the fault-free executions if $H \geq 1$ [83].
- As stated before, the values of S control the diversity of the views. In practice it means,

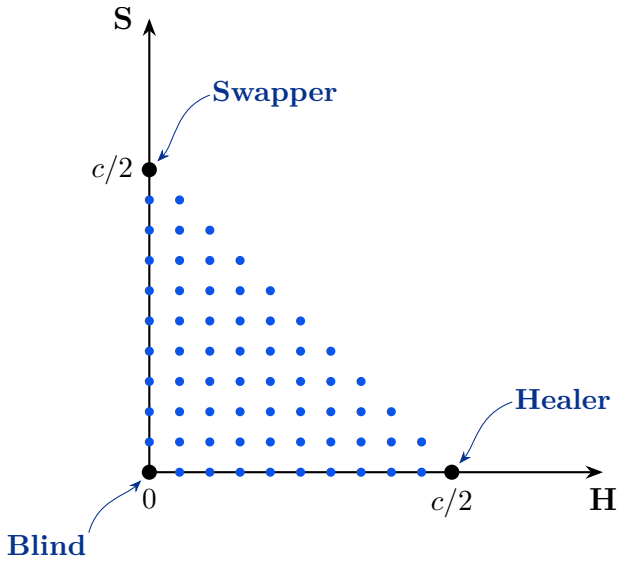


Figure C.2: The triangle of gossip protocols. In this example $c = 20$. Extreme settings are marked with big, black points and all possible states in between them with small, blue points.

that the probability of including the cache entry received during the communication phase increases with the increasing value of S, with a side effect of removing more and more unique addresses (comparing both caches of peers participating in the messages' exchange).

Each possible configuration enforces some trade-offs and there is no ideal setting [83].

The last parameter is the synchronisation time T. A conservative setting, commonly used in simulations is 10 seconds. The value of T affects the load of each peer during the protocol's execution — the lower the value, the higher the load.

Graph metrics for the emerging P2P networks are usually computed in a simulated environment. The simulation is executed in the synchronised fashion with the synchronisation time set to T. In each step, every node is selected once — in random order — to initiate a cache-exchange. Such context allows defining a **convergence time**, as a time required by the network to reach the state of equilibrium (from the global point of view) from its initial configuration. Regardless of the initialisation method and the parameters' settings, all networks tested in [83] converge in less than 50 simulation steps. All the properties, metrics and values discussed further will refer to a situation, when the network is already in the state of equilibrium.

Although, the distribution of local graph metrics is not uniform among the peers, it does not create a permanent setting. The situation changes at the each synchronisation cycle of the protocol. Therefore, if there are any bottlenecks in the network, "they will not be associated with the same nodes all the time" [83]. Thus, this improves the robustness and provides sort of load balancing.

Existence of bottlenecks is determined by the in-degree distribution (see Definition 2.2.2). The higher this value is for a given node, the more incoming connections are expected (precisely d/c, where d is the in-degree and c is the maximum cache size). The discussed metric tracked over time for a single node has a similar distribution like the one computed for all the peers at any simulation step, therefore the value is dynamically changing and does not

remain static. In the extreme configurations, only the swapper has the normal distribution of the nodes' in-degree with the average at the maximum cache size. For two others, the distribution is skewed in the direction of lower values.

Usually, the path statistics and the clustering coefficient in the case of the P2P protocol analysis are measured on the underlying graph (see Definition 2.1.3). A small average path length (see Definition 2.2.9) is required to consider a concrete configuration of the gossip protocol as scalable. For all settings, the measured values of this metric are reported in [83] as very low, with the protocols having large S values to be the closest to the random graph (see Section 2.3).

The last discussed metric is the clustering coefficient (see Definition 2.2.12). In the context of P2P protocols, "a high clustering coefficient has potentially damaging effects on both information dissemination (by increasing the number of redundant messages) and also on the self-healing capacity by weakening the connection of a cluster to the rest of the graph thereby increasing the probability of partitioning" [83]. Yet, for the distributed Agent-based Evolutionary Algorithms (AEAs) the redundancy of the messages is a desired property of the protocol. As reported in [83], the clustering is mainly controlled by H. Its high values result in significant clustering, which are far from the random graph. This is due to the high similarity of the cache occurring right after the exchange and integration of the cache entries. For the high values of S, this statistic is close to random, because S controls the diversity of the caches.

C.3 Chating faults and "malicious gossip"

P2P protocols, like DGVCS's, have to cope with the malicious behaviour of their participants. Although in this case, such actions are not driven by the incentives of any kind, but it is rather a problem associated with the anonymity (as described in Section 3.1.3). Byzantine attacks on the network are aimed to bias the peer selection process [84] or break the connectivity between the nodes. On of the key requirements making gossip protocols effective, namely the *push* part of the communication, is also the cause of the problem in case when the contents of the messages are altered, because it aids active spreading of the falsified data (what will be visible further).

C.3.1 Attack models

This situation is illustrated on Figure C.3. If at any time during the execution a group of nodes contact in a sequence a single, designed peer, then his view (or cache) will be (with a high probability) fully filled with their descriptors. If some other member of the system will not contact the mentioned node after that, then he will not be able to reconnect with the network. Yet, even if this happens, there is not guarantee that the node will stay connected to the network for long. It all depends on the protocol parameters, as it is highly probable that some malicious descriptors will be retained in the cache and the peer might simply choose one of the members of a malicious group for the subsequent cache exchange. As a side effect, a node which helped with the partial recovery of the view (with a high chance) will also integrate the malicious data into his cache, experiencing the same risk as described above.

Figure C.4 presents one of possible variants of the attack executed in a full scale. During the time when all members of the network follow the protocol (Figure C.4a), all the benefits

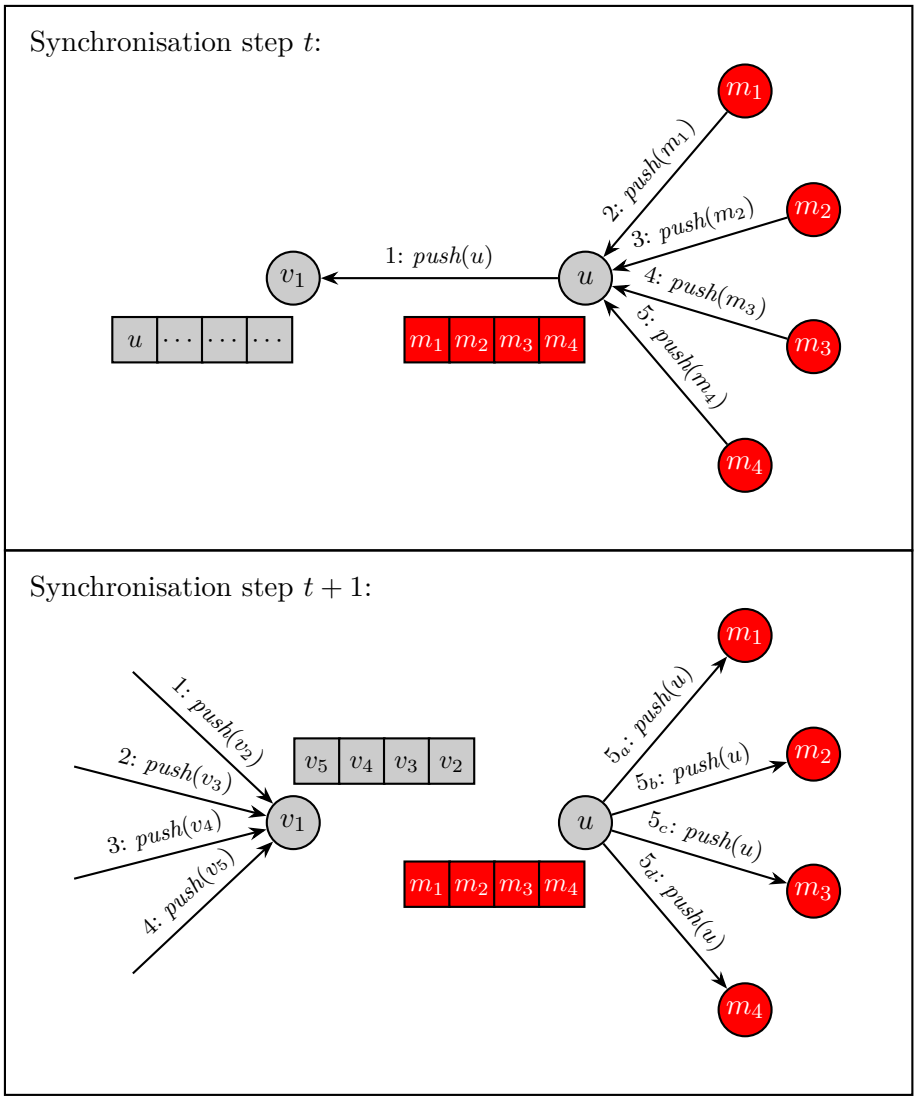
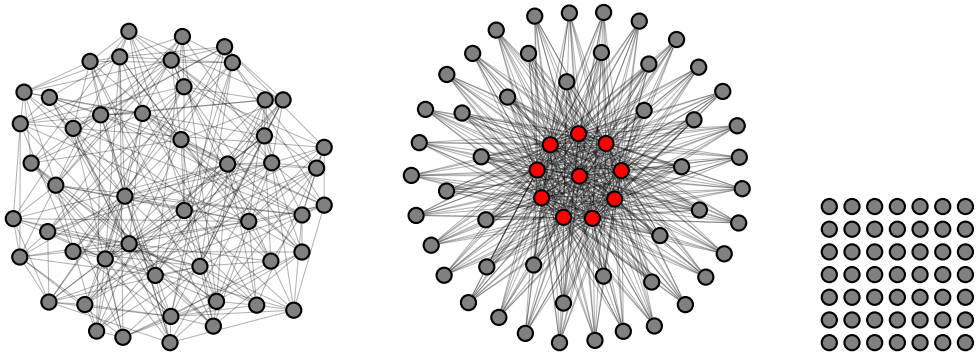


Figure C.3: The cache takeover (i.e. view poisoning) attack. Here the situation is greatly simplified: the maximum size of the cache is small (only four cache entries) and the execution is synchronous. After a current target of the attack is chosen (node u), cheating peers (from m_1 to m_4 in this example) start to flood the u's local view (messages from $2-5$) with the addresses from their own group. Upon the success, the content of the cache of the targeted node is filled with descriptors leading only to the malicious group. Additionally, possibly cutting of u from the rest of the network. If by some chance, the attacked peer managed to send its descriptor to the network before or during the attack, then it is possible that his view may be partially restored to the correct state by the incoming connection. On the illustration, the ID of u was "pushed" to v_1 at the beginning of the synchronisation phase (message number one), unfortunately it was replaced by other, fresher entries during the later steps of the execution (see the bottom figure).



(a) (b) (c)

Figure C.4: An example of the "hub attack". The network consists of 49 nodes, with the cache size limit set to 10. Figure (a) presents a snapshot of the network before the attack, (b) the culmination point of the activities of 10 cooperating, malicious users. They took the control over the network by filling the caches of the other peers with entries pointing to the malicious group — forming a hub-based overlay. The state of the network after malicious group left the system is depicted on Figure (c) — a set of fully disconnected nodes.

of gossiping described before are provided and all properties of the connection graph are preserved.

At some point, a group of colluding nodes joins the system — the number of the attackers does not have to be big and generally the "maximum cache size (c)" of them is enough to cause the problem. Instead of the normal execution of the protocol, they pursue their own agenda by sending the descriptors of the malicious group members. This in turn causes increasing pollution of the network with their addresses, with the additional and unaware help of the honest nodes. The more malicious descriptors are present, the higher is the chance that some node will contact an attacker facilitating further spread of this harmful data. All this, finally leads to a situation depicted on Figure C.4b.

When such state of the execution is reached, malicious users have the full control of the information flow within the network, forming a **hub** — therefore "the hub-attack" [84]. Yet, it is not the only issue: if at any time after that all members of the attacking group decide to leave the system, the rest of the nodes will become instantaneously disconnected (see Figure C.4c).

References

[1] E. H. L. Aarts, A. E. Eiben, and K. M. van Hee. *A General Theory of Genetic Algorithms*. Computing science notes. Univ. of Technology, Dep. of Mathematics and Computing Science, Computing Science Section, 1989.

[2] E. Alba and B. Dorronsoro. *Cellular Genetic Algorithms*. Springer Publishing Company, Incorporated, 1st edition, 2008.

[3] E. Alba, B. Dorronsoro, M. Giacobini, and M. Tomassini. *Handbook of Bioinspired Algorithms and Applications*, chapter Decentralized Cellular Evolutionary Algorithms, pages 103–120. CRC Press, 2006.

[4] E. Alba, A. J. Nebro, and J. M. Troya. Heterogeneous computing and parallel genetic algorithms. *J. Parallel Distrib. Comput.*, 62(9):1362–1385, 2002.

[5] E. Alba and M. Tomassini. Parallelism and Evolutionary Elgorithms. *IEEE Trans. Evol. Comput.*, 6(5):443–462, Oct. 2002.

[6] Amazon. Amazon Elastic Compute Cloud (Amazon EC2). `http://aws.amazon.com/ec2/`.

[7] D. P. Anderson. BOINC: A System for Public-Resource Computing and Storage. In *Fifth IEEE/ACM Int. Workshop on Grid Computing*, pages 4–10. IEEE, 2004.

[8] D. P. Anderson. Volunteer Computing: The Ultimate Cloud. *Crossroads*, 16(3):7–10, Mar. 2010.

[9] D. P. Anderson, J. Cobb, E. Korpela, M. Lebofsky, and D. Werthimer. SETI@home: an experiment in public-resource computing. *Communications of the ACM*, 45(11):56–61, Nov. 2002.

[10] D. P. Anderson, E. Korpela, and R. Walton. High-Performance Task Distribution for Volunteer Computing. In *Proc. of the 1st Int. Conf. on e-Science and Grid Computing*, E-SCIENCE'05, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.

[11] A. Avizienis, J.-C. Laprie, B. Randell, and C. E. Landwehr. Basic Concepts and Taxonomy of Dependable and Secure Computing. *IEEE Transactions on Dependable and Secure Computing*, 1:11–33, 2004.

[12] T. Bäck. Selective Pressure in Evolutionary Algorithms: A Characterization of Selection Mechanisms. In *Proc. of the 1st IEEE Conf. on Evolutionary Computation*, pages 57–62. IEEE Press, 1994.

[13] T. Bäck. Generalized Convergence Models for Tournament and (μ, λ)-Selection. In *Proc. of the 6th Int. Conf. on Genetic Algorithms*, pages 2–8, San Francisco, CA, USA, 1995. Morgan Kaufmann Publishers Inc.

[14] T. Bäck. Order Statistics for Convergence Velocity Analysis of Simplified Evolutionary Algorithms. volume 3 of *Foundations of Genetic Algorithms*, pages 91–102. Elsevier, 1995.

[15] M. Bakkaloglu, J. J. Wylie, C. Wang, and G. R. Ganger. On correlated failures in survivable storage systems. Technical Report MU-CS-02-129, Carnegie Mellon University, May 2002.

[16] A. Bakker and M. van Steen. Puppetcast: A secure peer sampling protocol. In *European Conf. on Computer Network Defense, 2008. EC2ND 2008*, pages 3–10, 2008.

[17] B. Bállobas and O. M. Riordan. *Mathematical results on scale-free random graphs*, pages 1–37. Wiley, 2003.

[18] A.-L. Barabási and R. Albert. Emergence of scaling in random networks. *Science*, 286(Oct.):509–512, 1999.

[19] H. E. C. Barrera, E. E. R. Rosero, and M. J. V. Cano. Desktop Grids and Volunteer Computing Systems. In N. Preve, editor, *Computational and Data Grids: Principles, Applications, and Design*, pages 1–30. IGI Global, Sept. 2011.

[20] R. Bhagwan, S. Savage, and G. Voelker. Understanding Availability. In *Proc. of The 2nd Int. Workshop on Peer-to-Peer Systems (IPTPS'03)*, 2003.

[21] M. Biazzini and A. Montresor. p2poem: Function optimization in P2P networks. *Peer-to-Peer Networking and Applications*, 6(2):213–232, 2013.

[22] T. Blickle and L. Thiele. A Comparison of Selection Schemes used in Genetic Algorithms. Technical report, Gloriastrasse 35, CH-8092 Zurich: Swiss Federal Institute of Technology (ETH) Zurich, Computer Engineering and Communications Networks Lab (TIK, 1995.

[23] E. Bortnikov, M. Gurevich, I. Keidar, G. Kliot, and A. Shraer. Brahms: Byzantine resilient random membership sampling. *Computer Networks*, 2009.

[24] B. Bánhelyi, M. Biazzini, A. Montresor, and M. Jelasity. Peer-to-Peer Optimization in Large Unreliable Networks with Branch-and-Bound and Particle Swarms. In M. Giacobini, A. Brabazon, S. Cagnoni, G. Di Caro, A. Ekárt, A. Esparcia-Alcázar, M. Farooq, A. Fink, and P. Machado, editors, *Applications of Evolutionary Computing*, volume 5484 of *Lecture Notes in Computer Science*, pages 87–92. Springer Berlin Heidelberg, 2009.

[25] L. Canon, E. Jeannot, and J. Weissman. A Scheduling and Certification Algorithm for Defeating Collusion in Desktop Grids. In *31st Int. Conf. on Distributed Computing Systems (ICDCS)*, pages 343–352, June 2011.

[26] E. Cantú-Paz. A Survey of Parallel Genetic Algorithms. *Calculateurs Paralleles, Reseaux et Systems Repartis*, 10, 1998.

[27] E. Cantú-Paz. Topologies, Migration Rates, and Multi-Population Parallel Genetic Algorithms. In W. Banzhaf, J. M. Daida, A. E. Eiben, M. H. Garzon, V. Honavar, M. J. Jakiela, and R. E. Smith, editors, *Proc. of the Conf. on Genetic and Evolutionary Computation (GECCO)*, pages 91–98. Morgan Kaufmann, 1999.

[28] E. Cantú-Paz. *Efficient and Accurate Parallel Genetic Algorithms*. Kluwer Academic Publishers, Norwell, MA, USA, 2000.

[29] U. K. Chakraborty, K. Deb, and M. Chakraborty. Analysis of Selection Algorithms: A Markov Chain Approach. *Evol. Comput.*, 4(2):133–167, June 1996.

[30] Z. Chen, G. E. Fagg, E. Gabriel, J. Langou, T. Angskun, G. Bosilca, and J. Dongarra. Fault Tolerant High Performance Computing by a Coding Approach. In *Proc. of the 10th ACM SIGPLAN Symp. on Principles and Practice of Parallel Programming*, PPoPP'05, pages 213–223, New York, NY, USA, 2005. ACM.

[31] I. Clarke, O. Sandberg, B. Wiley, and T. W. Hong. Freenet: A distributed anonymous information storage and retrieval system, 1999.

[32] N. Cole, T. J. Desell, D. L. Gonzalez, F. F. de Vega, M. Magdon-Ismail, H. J. Newberg, B. K. Szymanski, and C. A. Varela. Evolutionary Algorithms on Volunteer Computing Platforms: The MilkyWay@Home Project. In F. F. de Vega and E. Cantú-Paz, editors, *Parallel and Distributed Computational Intelligence*, volume 269 of *Studies in Computational Intelligence*, pages 63–90. Springer, 2010.

[33] T. H. Cormen, C. Stein, R. L. Rivest, and C. E. Leiserson. *Introduction to Algorithms*. McGraw-Hill Higher Education, 2nd edition, 2001.

[34] G. Coulouris, J. Dollimore, T. Kindberg, and G. Blair. *Distributed Systems: Concepts and Design*. Addison-Wesley Publishing Company, USA, 5th edition, 2011.

[35] Cumulus Project. `http://www.cumulus-project.eu/`.

[36] C. Darwin. *The Origin of Species*. John Murray, 1859.

[37] F. F. de Vega. A Fault Tolerant Optimization Algorithm based on Evolutionary Computation. In *Proc. of the Int. Conf. on Dependability of Computer Systems (DEPCOS-RELCOMEX'06)*, pages 335–342, Washington, DC, USA, 2006. IEEE Computer Society.

[38] P. Del Moral and A. Doucet. Particle methods: An introduction with applications, 2009.

[39] P. Del Moral, L. Kallel, and J. Rowe. Modeling Genetic Algorithms with Interacting Particle Systems. In *In Theoretical Aspects of Evolutionary Computing*, pages 10–67. Springer Verlag, 2001.

[40] T. Desell, M. Magdon-Ismail, B. Szymanski, C. A. Varela, B. A. Willett, M. Arsenault, and H. Newberg. Evolving N-Body Simulations to Determine the Origin and Structure of the Milky Way Galaxy's Halo Using Volunteer Computing. In *IEEE Int. Symp. on Parallel and Distributed Processing, Workshops and Phd Forum (IPDPSW)*, pages 1888–1895, May 2011.

[41] T. Desell, B. Szymanski, and C. Varela. An Asynchronous Hybrid Genetic-simplex Search for Modeling the Milky Way Galaxy Using Volunteer Computing. In *Proc. of the 10th Annu. Conf. on Genetic and Evolutionary Computation*, GECCO'08, pages 921–928, New York, NY, USA, 2008. ACM.

[42] T. J. Desell, M. Magdon-Ismail, B. K. Szymanski, C. A. Varela, H. J. Newberg, and D. P. Anderson. Validating Evolutionary Algorithms on Volunteer Computing Grids. In F. Eliassen and R. Kapitza, editors, *DAIS*, volume 6115 of *Lecture Notes in Computer Science*, pages 29–41, 2010.

[43] Distributed.net. `http://www.distributed.net/`, 1997.

[44] W. Du, J. Jia, M. Mangal, and M. Murugesan. Uncheatable grid computing. In *Proc. of 24th Int. Conf. on Distributed Computing Systems*, pages 4–11, 2004.

[45] R. Durrett. *Random Graph Dynamics*. Cambridge University Press, 2006. Cambridge Books Online.

[46] A. Dutot, F. Guinand, D. Olivier, and Y. Pigné. GraphStream: A Tool for bridging the gap between Complex Systems and Dynamic Graphs. In *Emergent Properties in Natural and Artificial Complex Systems. Satellite Conf. within the 4th European Conf. on Complex Systems (ECCS'2007)*, Dresden, Allemagne, 2007-10. ANR SARAH.

[47] J. Eberspächer and R. Schollmeier. First and Second Generation of Peer-to-Peer Systems. In R. Steinmetz and K. Wehrle, editors, *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*, pages 35–56. Springer, 2005.

[48] A. E. Eiben and J. E. Smith. *Introduction to Evolutionary Computing*. SpringerVerlag, 2003.

[49] E. N. M. Elnozahy, L. Alvisi, Y.-M. Wang, and D. B. Johnson. A Survey of Rollback-recovery Protocols in Message-passing Systems. *ACM Comput. Surv.*, 34(3):375–408, Sept. 2002.

[50] P. Erdös and A. Rényi. On random graphs. *Publicationes Mathematicae (Debrecen)*, 6:290–297, 1959.

[51] Eucalyptus. `https://www.eucalyptus.com/`.

[52] Facebook. `http://www.facebook.com`.

[53] Failure Trace Archive (FTA). `http://fta.scem.uws.edu.au`.

[54] F. Fernandez, M. Tomassini, and L. Vanneschi. Saving computational effort in genetic programming by means of plagues. In *The 2003 Congr. on Evolutionary Computation (CEC'03)*, volume 3, pages 2042–2049, Dec. 2003.

[55] Flickr. `http://www.flickr.com`.

[56] Folding@home. `http://folding.stanford.edu`.

[57] A. Fronczak, P. Fronczak, and J. A. Hołyst. Average path length in random networks. *Phys. Rev. E*, 70:056110, Nov. 2004.

[58] C. Germain and D. Monnier-Ragaigne. Grid Result Checking. In *Proc. of the 2nd Conf. on Computing frontiers*, pages 87–96, Ischia, Italy, May 2005. ACM Press.

[59] M. Giacobini, M. Tomassini, and A. Tettamanzi. Takeover Time Curves in Random and Small-world Structured Populations. In *Proc. of the 7th Annu. on Conf. on Genetic and Evolutionary Computation*, GECCO'05, pages 1333–1340, New York, NY, USA, 2005. ACM.

[60] GIMPS. Great Internet Mersenne Prime Search. `http://www.mersenne.org/`, 1996.

[61] GoGrid. `http://www.gogrid.com/`.

[62] D. E. Goldberg. *Genetic Algorithms in Search, Optimization and Machine Learning.* Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 1st edition, 1989.

[63] D. E. Goldberg and K. Deb. A Comparative Analysis of Selection Schemes Used in Genetic Algorithms. In G. J. E. Rawlins, editor, *Foundations of Genetic Algorithms*, pages 69–93. San Francisco, CA: Morgan Kaufmann, 1991.

[64] D. L. González, F. F. de Vega, and H. Casanova. Characterizing Fault Tolerance in Genetic Programming. In *Proc. of the 2009 workshop on Bio-inspired algorithms for distributed systems (BADS'09)*, pages 1–10, New York, NY, USA, 2009. ACM.

[65] D. L. González, F. F. de Vega, and H. Casanova. Characterizing fault tolerance in genetic programming. *Future Generation Computer Systems*, 26(6):847–856, June 2010.

[66] D. L. Gonzalez, F. F. de Vega, L. Trujillo, G. Olague, L. Araujo, P. Castillo, J. J. Merelo, and K. Sharman. Increasing GP Computing Power for Free via Desktop GRID Computing and Virtualization. In *17th Euromicro Int. Conf. on Parallel, Distributed and Network-based Processing*, pages 419–423, Feb. 2009.

[67] Google. Google App Engine: Platform as a Service — Google Cloud Platform. `https://cloud.google.com/appengine/docs`.

[68] Google. Google Gmail. `https://www.gmail.com/`.

[69] Grid'5000. `https://www.grid5000.fr/`.

[70] S. Guha, N. Daswani, and R. Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. In *Proc. of The 5th Int. Workshop on Peer-to-Peer Systems (IPTPS'06)*, Feb. 2006.

[71] P. Hage and H. F. *Structural Models in Anthropology.* Cambridge University Press, 1984. Cambridge Books Online.

[72] R. Hauser and R. Männer. Implementation of Standard Genetic Algorithm on MIMD Machines. pages 504–513, 1994.

[73] J. He and X. Yao. Drift analysis and average time complexity of evolutionary algorithms. *Artificial Intelligence*, 127(1):57–85, 2001.

[74] J. He and X. Yao. A study of drift analysis for estimating computation time of evolutionary algorithms. *Natural Computing*, 3(1):21–35, 2004.

[75] J. I. Hidalgo, J. Lanchares, F. Fernández de Vega, and D. Lombrana. Is the island model fault tolerant? In *GECCO '07: Proc. of the 2007 GECCO conference companion on Genetic and Evolutionary Computation*, pages 2737–2744, London, United Kingdom, July 7–11 2007. ACM.

[76] HTCondor. `http://research.cs.wisc.edu/htcondor/`.

[77] K. Hwang, J. Dongarra, and G. C. Fox. *Distributed and Cloud Computing: From Parallel Processing to the Internet of Things.* Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1st edition, 2011.

[78] A. Iosup, M. Jan, O. Sonmez, and D. H. J. Epema. On the Dynamic Resource Availability in Grids. In *8th IEEE/ACM Int. Conf. on Grid Computing*, Sept. 2007.

[79] B. Javadi, D. Kondo, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling the Comparison of Failure Measurements and Models of Distributed Systems. *Journal of Parallel and Distributed Computing*, 73:1208–1223, Aug. 2013.

[80] B. Javadi, D. Kondo, J. Vincent, and D. P. Anderson. Mining for Statistical Availability Models in Large-Scale Distributed Systems: An Empirical Study of SETI@home. In *17th IEEE/ACM Int. Symp. on Modelling, Analysis and Simulation of Computer and Telecommunication Systems (MASCOTS)*, Sept. 2009.

[81] B. Javadi, D. Kondo, J.-m. Vincent, and D. Anderson. Discovering Statistical Models of Availability in Large Distributed Systems: An Empirical Study of SETI@home. *Parallel and Distributed Systems, IEEE Transactions on*, 22(11):1896–1903, Nov. 2011.

[82] M. Jelasity and M. van Steen. Large-Scale Newscast Computing on the Internet. Technical Report IR-503, Vrije Universiteit Amsterdam, Department of Computer Science, Amsterdam, The Netherlands, Oct. 2002.

[83] M. Jelasity, S. Voulgaris, R. Guerraoui, A.-M. Kermarrec, and M. van Steen. Gossip-based Peer Sampling. *ACM Trans. Comput. Syst.*, 25(3), 2007.

[84] G. P. Jesi, A. Montresor, and M. van Steen. Secure peer sampling. *Computer Networks*, 54:2086–2098, 2010.

[85] J. L. Jiménez Laredo. *Peer-to-Peer Evolutionary Computation: A Study of Viability.* PhD thesis, Universidad de Granada, Apr. 2010.

[86] J. L. Jiménez Laredo, P. Bouvry, D. L. González, F. F. de Vega, M. G. Arenas, J. J. Merelo, and C. M. Fernandes. Designing robust volunteer-based evolutionary algorithms. *Genetic Programming and Evolvable Machines*, 15(3):221–244, 2014.

[87] J. L. Jiménez Laredo, P. A. Castillo, A. M. Mora, and J. J. Merelo. Exploring population structures for locally concurrent and massively parallel Evolutionary Algorithms. In *IEEE Congr. on Evolutionary Computation, 2008. CEC 2008. (IEEE World Congr. on Computational Intelligence)*, pages 2605–2612, June 2008.

[88] J. L. Jiménez Laredo, A. E. Eiben, M. van Steen, and J. J. Merelo. EvAg: A Scalable Peer-to-peer Evolutionary Algorithm. *Genetic Programming and Evolvable Machines*, 11(2):227–246, June 2010.

[89] H. Johansen, A. Allavena, and R. van Renesse. Fireflies: Scalable Support for Intrusion-tolerant Network Overlays. *SIGOPS Oper. Syst. Rev.*, 40(4):3–13, Apr. 2006.

[90] D. Kondo, F. Araujo, P. Malecot, P. Domingues, L. Silva, G. Fedak, and F. Cappello. Characterizing Result Errors in Internet Desktop Grids. *Euro-Par 2007 Parallel Processing*, pages 361–371, 2007.

[91] D. Kondo, G. Fedak, F. Cappello, A. A. Chien, and H. Casanova. Characterizing Resource Availability in Enterprise Desktop Grids. *Future Gener. Comput. Syst.*, 23(7):888–903, Aug. 2007.

[92] D. Kondo, B. Javadi, A. Iosup, and D. Epema. The Failure Trace Archive: Enabling Comparative Analysis of Failures in Diverse Distributed Systems. In *Proc. of the 2010 10th IEEE/ACM Int. Conf. on Cluster, Cloud and Grid Computing*, CCGRID'10, pages 398–407, Washington, DC, USA, 2010. IEEE Computer Society.

[93] P. Kouchakpour, A. Zaknich, and T. Bräunl. Dynamic Population Variation in Genetic Programming. *Inf. Sci.*, 179(8):1078–1091, Mar. 2009.

[94] Y.-K. Kwok. *The Handbook of Computer Networks*, volume 3, chapter Incentive Issues in Peer-to-Peer Systems, pages 168–188. John Wiley & Sons, Dec. 2007.

[95] Y.-K. R. Kwok. *Peer-to-Peer Computing: Applications, Architecture, Protocols, and Challenges*. CRC Press, 2012.

[96] J. L. J. Laredo, P. Bouvry, S. Mostaghim, and J. J. Merelo-Guervós. Validating a Peer-to-Peer Evolutionary Algorithm. In C. Chio, A. Agapitos, S. Cagnoni, C. Cotta, F. F. de Vega, G. A. Caro, R. Drechsler, A. Ekárt, A. I. Esparcia-Alcázar, M. Farooq, W. B. Langdon, J. J. Merelo-Guervós, M. Preuss, H. Richter, S. Silva, A. Simoes, G. Squillero, E. Tarantino, A. G. B. Tettamanzi, J. Togelius, N. Urquhart, A. S. Uyar, and G. N. Yannakakis, editors, *Applications of Evolutionary Computation*, volume 7248 of *Lecture Notes in Computer Science*, pages 436–445. Springer Berlin Heidelberg, 2012.

[97] J. L. J. Laredo, P. A. Castillo, A. M. Mora, J. J. Merelo, and C. Fernandes. Resilience to churn of a Peer-to-Peer Evolutionary Algorithm. *Int. J. High Perform. Syst. Archit.*, 1(4):260–268, Mar. 2008.

[98] J. Lässig and D. Sudholt. General scheme for analyzing running times of parallel evolutionary algorithms. *Parallel Problem Solving from Nature, PPSN XI*, pages 234–243, 2011.

[99] M. Litzkow, M. Livny, and M. Mutka. Condor — a Hunter of Idle Workstations. In *The 8th Int. Conf. on Distributed Computing Systems*, pages 104–111. IEEE Comput. Soc. Press, 1988.

[100] E. K. Lua, J. Crowcroft, M. Pias, R. Sharma, and L. Steven. A survey and comparison of peer-to-peer overlay network schemes. *IEEE Communications Survey and Tutorial*, (Mar.), 2004.

[101] S. Luke, G. C. Balan, and L. Panait. Population implosion in Genetic Programming. In *Proc. of the 2003 Int. Conf. on Genetic and Evolutionary Computation: PartII*, GECCO'03, pages 1729–1739, Berlin, Heidelberg, 2003. Springer-Verlag.

[102] D. Malkhi, M. Naor, and D. Ratajczak. Viceroy : A Scalable and Dynamic Emulation of the Butterfly. In *Proc. of the of the ACM PODC02*, pages 183–192, 2002.

[103] P. Maymounkov and D. Mazieres. Kademlia: A Peer-to-peer Information System Based on the XOR Metric. In *Proc. of the 1st Int. Workshop Peer-to-Peer Systems (IPTPS)*, pages 53–65, 2002.

[104] N. Melab, S. Cahon, and E. Talbi. Grid computing for parallel bioinspired algorithms. *Journal of Parallel and Distributed Computing*, 66(8):1052–1061, Aug. 2006.

[105] Z. Michalewicz. *Genetic Algorithms + Data Structures = Evolution Programs*. Springer-Verlag, London, UK, UK, 3 edition, 1996.

[106] Microsoft. Microsfot Azure. `http://azure.microsoft.com/`.

[107] Y. M. Minsky and F. B. Schneider. Tolerating Malicious Gossip. *Distributed Computing*, 16(1):49–68, 2003.

[108] D. Molnar. The SETI@Home Problem. `http://www.acm.org/crossroads/columns/onpatrol/september2000.html`, Sept. 2000.

[109] E. Montero and M.-C. Riff. On-the-fly calibrating strategies for evolutionary algorithms. *Information Sciences*, 181(3):552–566, 2011.

[110] R. Moore. *Cybercrime: Investigating High-Technology Computer Crime*. LexisNexis/Matthew Bender, 2005.

[111] A. Morales-Reyes, E. F. Stefatos, A. T. Erdogan, and T. Arslan. Towards Fault-Tolerant Systems based on Adaptive Cellular Genetic Algorithms. In *Proc. of the 2008 NASA/ESA Conf. on Adaptive Hardware and Systems (AHS'08)*, pages 398–405, Noordwijk, The Netherlands, June 22-25 2008. IEEE Computer Society.

[112] J. Muszyński, S. Varrette, and P. Bouvry. Expected Running Time of Parallel Evolutionary Algorithms on Unimodal Pseudo-Boolean Functions over Small-World Networks. In *Proc. of the IEEE Congr. on Evolutionary Computation (CEC'2013)*, pages 2588–2594, Cancún, Mexico, June 2013. IEEE.

[113] J. Muszyński, S. Varrette, and P. Bouvry. On the Resilience of the Newscast Protocol in the Presence of Cheaters. In *2014 Grande Region Security and Reliability Day (GRSRD 2014)*, Saarbrücken, Germany, Mar. 2014.

[114] J. Muszyński, S. Varrette, P. Bouvry, F. Seredyński, and S. U. Khan. Convergence Analysis of Evolutionary Algorithms in the Presence of Crash-Faults and Cheaters. *Int. Journal. of Computers and Mathematics with Applications (CAMWA)*, 64(12):3805–3819, Dec. 2012.

[115] J. Muszyński, S. Varrette, J. L. Jiménez Laredo, and P. Bouvry. Analysis of the Data Flow in the Newscast Protocol for Possible Vulnerabilities. In *Proc. of Int. Conf. on Cryptography and Security System (CSS'14)*, volume 448 of *Communications in Computer and Information Sciences (CCIS)*, pages 89–99, Lublin, Poland, Sept. 2014. Springer.

[116] J. Muszyński, S. Varrette, J. L. Jiménez Laredo, and P. Bouvry. Exploiting the Hard-wired Vulnerabilities of Newscast via Connectivity-splitting Attack. In *Proc. of the IEEE Int. Conf. on Network and System Security (NSS'2014)*, volume 8792 of *LNCS*, pages 152–165, Xi'an, China, Oct. 2014. Springer Verlag. **Best Student Paper Award**.

[117] J. Muszyński, S. Varrette, J. L. Jiménez Laredo, B. Dorronsoro, and P. Bouvry. Convergence of Distributed Cellular Evolutionary Algorithms in Presence of Crash Faults and Cheaters. In *Proc. of the Int. Conf. on Metaheuristics and Nature Inspired Computing (META'12)*, Sousse, Tunisia, Oct. 27–31 2012.

[118] Napster. `http://www.napster.com/`.

[119] M. Newman. *Networks: An Introduction*. Oxford University Press, Inc., New York, NY, USA, 2010.

[120] M. E. J. Newman. Assortative mixing in networks. *Phys. Rev. Lett.*, 89:208701, May 2002.

[121] M. E. J. Newman. The structure and function of complex networks. *SIAM review*, 45(2):167–256, Mar. 2003.

[122] M. E. J. Newman and D. J. Watts. Scaling and percolation in the small-world network model. *Physical Review E*, 87501, 1999.

[123] Nimbus. `http://www.nimbusproject.org/`.

[124] A. Oliner and J. Stearley. What Supercomputers Say: A Study of Five System Logs. In *Int. Symp. on Dependable Systems and Networks (DSN)*, June 2007.

[125] P. S. Oliveto, J. He, and X. Yao. Time complexity of evolutionary algorithms for combinatorial optimization: A decade of results. *Int. J. of Automation and Computing*, pages 100–106, 2007.

[126] OpenNebula. `http://opennebula.org/`.

[127] OpenQRM Enterprise. `http://www.openqrm-enterprise.com/`.

[128] OpenStack. `http://openstack.org/`.

[129] U.-M. OReilly, M. Wagy, and B. Hodjat. EC-Star: A Massive-Scale, Hub and Spoke, Distributed Genetic Programming System. In R. Riolo, E. Vladislavleva, M. D. Ritchie, and J. H. Moore, editors, *Genetic Programming Theory and Practice X*, Genetic and Evolutionary Computation, pages 73–85. Springer New York, 2013.

[130] J. Pang, J. Hendricks, A. Akella, B. Maggs, R. D. Prisco, and S. Seshan. Availability,usage, and deployment characteristics of the domain name system. In *Internet Measurement Conf. (IMC)*, Oct. 2004.

[131] J. A. Patel, I. Gupta, and N. Contractor. JetStream: Achieving Predictable Gossip Dissemination by Leveraging Social Network Principles. In *Network Computing and Applications, 2006. NCA 2006.*, pages 32–39, July 2006.

[132] Rackspace managed cloud. `http://www.rackspace.com/cloud`.

[133] S. Ratnasamy, P. Francis, M. Handley, S. Shenker, and R. Karp. A Scalable Content-Addressable Network. In *Proc. of the ACM SIGCOMM 2002 Conf. on Applications, Technologies, Architectures, and Protocols for Computer Communications*, pages 161–172, 2001.

[134] A. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location and routing for large-scale peer-to-peer systems. In *Proc. of the Middleware*, 2001.

[135] G. Rudolph. Convergence Analysis of Canonical Genetic Algorithms. *Trans. Neur. Netw.*, 5(1):96–101, Jan. 1994.

[136] G. Rudolph. Convergence of Evolutionary Algorithms in General Search Spaces. In *Int. Conf. on Evolutionary Computation*, pages 50–54. IEEE, 1996.

[137] G. Rudolph. Finite Markov Chain Results in Evolutionary Computation: A Tour D'Horizon. *Fundam. Inf.*, 35(1-4):67–89, Aug. 1998.

[138] G. Rudolph. Takeover Times and Probabilities of Non-Generational Selection Rules. In L. D. Whitley, D. E. Goldberg, E. Cantu-Paz, L. Spector, I. C. Parmee, and H.-G. Beyer, editors, *Proc. of the Conf. on Genetic and Evolutionary Computation (GECCO)*, pages 903–910. Morgan Kaufmann, 2000.

[139] Salesforce.com Inc. Salesfoce Platform. `http://www.salesforce.com`.

[140] L. F. G. Sarmenta. Sabotage-Tolerance Mechanisms for Volunteer Computing Systems. *Future Generation Computer Systems*, 18(4):561–572, Mar. 2002.

[141] S. Saroiu, K. P. Gummadi, and S. D. Gribble. Measuring and Analyzing the Characteristics of Napster and Gnutella Hosts. *Multimedia Syst.*, 9(2):170–184, Aug. 2003.

[142] B. Schroeder and G. A. Gibson. Disk failures in the real world: What does an MTTF of 1,000,000 hours mean to you? In *5th Usenix Conf. on File and Storage Technologies (FAST)*, Feb. 2007.

[143] I. Scriven, D. Ireland, A. Lewis, S. Mostaghim, and J. Branke. Asynchronous multiple objective particle swarm optimisation in unreliable distributed environments. In *IEEE Congr. on Evolutionary Computation*, pages 2481–2486. IEEE, 2008.

[144] I. Scriven, A. Lewis, D. Ireland, and J. Lu. Decentralised distributed multiple objective Particle Swarm Optimisation using peer to peer networks. In *IEEE Congr. on Evolutionary Computation, 2008. CEC 2008. (IEEE World Congr. on Computational Intelligence)*, pages 2925–2928, June 2008.

[145] I. Scriven, A. Lewis, and S. Mostaghim. Dynamic search initialisation strategies for multi-objective optimisation in peer-to-peer networks. In *IEEE Congr. on Evolutionary Computation, 2009. CEC'09.*, pages 1515–1522, May 2009.

[146] S. Sen and J. Wang. Analyzing Peer-to-peer Traffic Across Large Networks. *IEEE/ACM Trans. Netw.*, 12(2):219–232, Apr. 2004.

[147] M. Sheikhalishahi, M. Devare, L. Grandinetti, and M. C. Incutti. A Complementary Approach to Grid and Cloud Distributed Computing Paradigms. In N. Preve, editor, *Computational and Data Grids: Principles, Applications, and Design*, pages 31–44. IGI Global, Sept. 2011.

[148] M. Sherr, B. T. Loo, and M. Blaze. Veracity: A Fully Decentralized Service for Securing Network Coordinate Systems. In *Proc. of the 7th Int. Conf. on Peer-to-peer Systems*, IPTPS'08, pages 15–15, Berkeley, CA, USA, 2008. USENIX Association.

[149] J. Shoch and J. Hupp. The "worm" programs — early experience with a distributed computation. *Communications of the ACM*, 25(3), 1982.

[150] J. Smith. On Replacement Strategies in Steady State Evolutionary Algorithms. *Evol. Comput.*, 15(1):29–59, 2007.

[151] R. Steinmetz and K. Wehrle, editors. *Peer-to-Peer Systems and Applications*, volume 3485 of *Lecture Notes in Computer Science*. Springer, 2005.

[152] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: a scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Transactions on Networking*, 11(1):17–32, Feb. 2003.

[153] D. Stutzbach and R. Rejaie. Understanding Churn in Peer-to-peer Networks. In *Proc. of the 6th ACM SIGCOMM Conf. on Internet Measurement*, IMC'06, pages 189–202, New York, NY, USA, 2006. ACM.

[154] A. S. Tanenbaum and M. van Steen. *Distributed Systems: Principles and Paradigms*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 2nd edition, 2006.

[155] The Gnutella Developer Forum (GDF). The annotated Gnutella protocol specification v0.4, 2001.

[156] The Gnutella Developer Forum (GDF). The Gnutella v0.6 protocol, 2001.

[157] D. Thierens and D. E. Goldberg. Mixing in Genetic Algorithms. In *Proc. of the 5th Int. Conf. on Genetic Algorithms*, pages 38–47, San Francisco, CA, USA, 1993. Morgan Kaufmann Publishers Inc.

[158] M. Tomassini. Parallel and Distributed Evolutionary Algorithms: A Review, 1999.

[159] M. Tomassini. *Spatially Structured Evolutionary Algorithms: Artificial Evolution in Space and Time*. Springer, 2005.

[160] M. Tomassini, L. Vanneschi, J. Cuendet, and F. Fernandez. A new technique for dynamic size populations in genetic programming. In *Congr. on Evolutionary Computation, 2004. CEC2004*, volume 1, pages 486–493, June 2004.

[161] L. Trujillo and G. Olague. Automated design of image operators that detect interest points. *Evolutionary computation*, 16(4):483–507, Jan. 2008.

[162] M. van Steen. *Graph Theory and Complex Networks: An Introduction*. Maarten van Steen, Apr. 2010.

[163] S. Varrette. *Sécurité des Architectures de Calcul Distribué: Authentification et Certification de Résultats*. PhD thesis, INP Grenoble and Université du Luxembourg, Sept. 2007. In French.

[164] S. Varrette, J. Muszyński, and P. Bouvry. Cheating impact on distributed Evolutionary Algorithms over BOINC computations. In *Proc. of the 19th Int. Conf. on Security and Intelligent Information Systems (SIIS 2011)*, Warsaw, Poland, June 13–14 2011. Extended Abstract.

[165] S. Varrette, J. Muszyński, and P. Bouvry. Hash function generation by means of Gene Expression Programming. In *Proc. of Int. Conf. on Cryptography and Security System (CSS'12)*, Kazimierz Dolny, Poland, Sept. 2012. Annales UMCS ser. Informatica.

[166] S. Varrette, M. Ostaszewski, and P. Bouvry. Nature inspired Algorithm-Based Fault Tolerance on Global Computing Platforms. Application to Symbolic Regression. In *Proc. of the Int. Conf. on Metaheuristics and Nature Inspired Computing (META'08)*, Hammamet, Tunisia, Oct. 29–31 2008.

[167] S. Varrette, V. Plugaru, M. Guzek, X. Besseron, and P. Bouvry. HPC Performance and Energy-Efficiency of the OpenStack Cloud Middleware. In *Proc. of the 43rd Int. Conf. on Parallel Processing (ICPP-2014), Heterogeneous and Unconventional Cluster Architectures and Applications Workshop (HUCAA'14)*, Minneapolis, MN, US, Sept. 2014. IEEE.

[168] S. Varrette, E. Tantar, and P. Bouvry. On the Resilience of [distributed] Evolutionary Algorithms against Cheaters in Global Computing Platforms. In *Proc. of the 14th Int. Workshop on Nature Inspired Distributed Computing (NIDISC 2011), part of the 25th IEEE/ACM Int. Parallel and Distributed Processing Symp. (IPDPS 2011)*, Anchorage (Alaska), USA, May 16–20 2011. IEEE Computer Society.

[169] F. Vega-Redondo. *Complex Social Networks*. Cambridge University Press, 2007.

[170] M. D. Vose. *The Simple Genetic Algorithm: Foundations and Theory*. MIT Press, Cambridge, MA, USA, 1998.

[171] M. D. Vose and G. Liepins. Punctuated equilibria in genetic search. *Complex Systems*, 5:31–44, 1991.

[172] A. Vosoughi, K. Bilal, S. U. Khan, N. Min-Allah, J. Li, N. Ghani, P. Bouvry, and S. Madani. A multidimensional robust greedy algorithm for resource path finding in large-scale distributed networks. In *Proc. of the 8th Int. Conf. on Frontiers of Information Technology*, FIT'10, pages 16:1–16:6, New York, NY, USA, 2010. ACM.

[173] S. Voulgaris, M. Jelasity, and M. van Steen. *Agents and Peer-to-Peer Computing*, volume 2872 of *Lecture Notes in Computer Science (LNCS)*, chapter A Robust and Scalable Peer-to-Peer Gossiping Protocol, pages 47–58. Springer Berlin / Heidelberg, 2004.

[174] H. Wasserman and M. Blum. Software Reliability via Run-Time Result-Checking. *Journal of the ACM*, 44(6):826–849, 1997.

[175] S. Wasserman, K. Faust, and D. Iacobucci. *Social Network Analysis : Methods and Applications (Structural Analysis in the Social Sciences)*. Cambridge University Press, Nov. 1994.

[176] D. J. Watts and S. H. Strogatz. Collective dynamics of 'small-world' networks. *Nature*, 393(6684):440–2, June 1998.

[177] I. Wegener. Theoretical Aspects of Evolutionary Algorithms. In *Proc. of the 28th Int. Colloquium on Automata, Languages and Programming*, ICALP'01, pages 64–78, London, UK, UK, 2001. Springer-Verlag.

[178] I. Wegener. Methods for the analysis of evolutionary algorithms on pseudo-boolean functions. *Evolutionary optimization*, (1990):1–21, 2003.

[179] J. M. Whitacre, R. A. Sarker, and Q. T. Pham. The Self-Organization of Interaction Networks for Nature-Inspired Optimization. *IEEE Trans. Evol. Comput.*, 12(2):220–230, Apr. 2008.

[180] XSEDE. Extreme Science and Engineering Discovery Environment. `https://www.xsede.org`.

[181] N. Yigitbasi, M. Gallet, D. Kondo, A. Iosup, and D. Epema. Analysis and Modeling of Time-Correlated Failures in Large-Scale Distributed Systems. In *8th IEEE/ACM Int. Conf. on Grid Computing*, Oct. 2010.

[182] B. Y. Zhao, L. Huang, J. Stribling, S. C. Rhea, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A Resilient Global-Scale Overlay for Service Deployment. *IEEE Journal on Selected Areas in Communications*, 22(1):41–53, Jan. 2004.

Acronyms

ABFT Algorithm-Based Fault Tolerance
AEA Agent-based Evolutionary Algorithm
BFS Breath-First Search
BOINC Berkeley Open Infrastructure for Network Computing
CAN Content Addressable Network
CC Cloud Computing
CEA Cellular Evolutionary Algorithm
CPU Central Processing Unit
DFS Depth-First Search
DGVCS Desktop Grid and Volunteer Computing System
DHT Distributed Hash Table
DNS Domain Name System
EA Evolutionary Algorithm
ES Evolutionary Strategy
EP Evolutionary Programming
EvAg Evolvable Agent Model
GA Genetic Algorithm
GC Global Computing
GP Genetic Programming
GPU Graphics Processing Unit
HPC High Performance Computing
HTC High Throughput Computing
IaaS Infrastructure as a Service
I/O Input/Output
IPS Interacting Particle System
IPv6 Internet Protocol version 6
ISP Internet Service Provider
LAN Local Area Network
OS Operating System
P2P Peer-to-Peer
PaaS Platform as a Service
PC Personal Computer
pEA parallel Evolutionary Algorithm
PSO Particle Swarm Optimisation
RIAA Recording Industry Association of America

SaaS	Software as a Service
TTL	Time-to-Live
UL	University of Luxembourg
VCS	Volunteer Computing System
VM	Virtual Machine
VT	Virtualisation Technology

List of Figures

List of Tables