Communication-Efficient Self-Stabilization

Dmitry Zinenko
Communication-Efficient Self-Stabilization

Research Thesis

Submitted in partial fulfillment of the requirements for the degree of Master of Science in Computer Science

Dmitry Zinenko

Submitted to the Senate of the Technion — Israel Institute of Technology
Tevet 5771 Haifa December 2010
The research thesis was done under the supervision of Prof. Shay Kutten from the Faculty of Industrial Engineering and Management in the Faculty of Computer Science.

First of all, I would like to thank my family and friends for their love and support for many years. Next, I thank my advisor Professor Shay Kutten for providing me with dozens of interesting research questions and relentless support throughout this work, even though the circumstances were often difficult. This research would not be possible without his guidance. I also thank Professors Chen Avin and Zvi Lotker from the Ben Gurion University of the Negev for hours of interesting discussions, especially during the early stages of this work.

My studies at the Technion provided me with opportunity to meet many exceptional people, and I would like to mention some of them. I thank Professors Shmuel Zaks and Hagit Attiya who introduced me to this field and helped through kind support and inspiring example. I would also like to thank Professor Assaf Schuster for his support and the opportunity to teach (and learn) many exciting things. I also thank Yardena Kolet for doing everything possible to make my job easier through never failing help and advice.

I dedicate this thesis to the memory of my grandfather, Nikolay Polyakov, whose keen mind and passion for knowledge have been, and will always remain, the main inspiration for me.

The generous financial support of the Technion and Israel Science Foundation is gratefully acknowledged.
# Contents

List of Figures .................................................. iii

Abstract .......................................................... 1

Abbreviations and Notations ...................................... 2

1 Introduction .................................................... 3
   1.1 Model ..................................................... 5
   1.2 Our Contribution ........................................ 7
   1.3 Related work ............................................ 8

2 The random branching process ................................ 12
   2.1 Proof of Theorem 2.1 .................................. 14
      2.1.1 Expected convergence speed ................ ..... 14
      2.1.2 Probability bounds for the convergence speed .... 17
   2.2 A generalization ........................................ 20

3 Spanning forest algorithm ..................................... 21
   3.1 Algorithm analysis .................................... 24

4 Low-bandwidth self-stabilizing reset ......................... 28
   4.1 The unison problem ................................... 29

5 Spanning tree algorithm .................................... 31
   5.1 Tree recoloring ....................................... 31
   5.2 Competitor detection and synchronization ............ 32
   5.3 Algorithm description and analysis .................. 34
6 Conclusion and directions for further work 38

Bibliography 41
List of Figures

1 Spanning forest algorithm, main loop .................................. 22
2 Spanning forest algorithm, request procedure .......................... 23
3 Spanning forest algorithm, confirmation procedure ..................... 23
4 Final spanning tree algorithm, main loop ................................. 35
5 Final spanning tree algorithm, reset handler procedure ............... 35
6 Final spanning tree algorithm, request procedure ........................ 36
7 Final spanning tree algorithm, confirmation procedure ............... 36
8 The effect of re-capturing on the convergence speed .................... 39
Abstract

In 1974, Dijkstra introduced the notion of self-stabilization in the context of distributed systems. He defined a system as self-stabilizing when “regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps.” Most self-stabilizing protocols rely on checking every neighbor of a node continuously to detect failures. Therefore, such protocols have a high communication cost, especially in dense graphs. Drawing inspiration from randomized gossip protocols, we investigate the potential effect of randomization on the communication efficiency of self-stabilizing protocols.

We study this approach in a complete graph, where the communication overhead seems to be the highest when one strives for protocols that are also fast. We present randomized low communication self-stabilizing algorithms for several major tasks, namely, spanning tree construction, distributed reset, and unison. The spanning tree algorithm sends a constant number of messages per node every round, and converges within $O(\log n)$ rounds with high probability. The reset and unison algorithms send only one message per node every round, and also converge within $O(\log n)$ rounds with high probability. This results in $O(n \log n)$ messages until convergence w.h.p. for all three algorithms, while all previously known self-stabilizing solutions for those problems have the message complexity of $O(n^2)$ in a complete graph.

Our reset and unison protocols can also be used (although with a different round complexity) in more general classes of synchronous networks. For example, in bounded degree networks their round complexity is $O(D + \log n)$ with high probability, where $D$ is the graph diameter, while still sending one message per node per round.
Abbreviations and Notations

The notation \( m.a \) represents field \( a \) in message \( m \). Sets and algorithms are represented by calligraphic capital letters, e.g., \( S, A \). Node variables appear in italics in the algorithms and in teletype script in the main text. Random variables use capital letters.

\begin{itemize}
  \item \( n \) — The number of nodes (processors).
  \item \( \Delta \) — Maximum degree of the network.
  \item \( d \) — Upper bound on the degree of the spanning tree.
  \item \( \log; e \text{ or } \exp \) — Natural logarithm; natural exponent.
  \item \( | \ldots | \) — Number of elements when applied to a set; absolute value when applied to a scalar.
  \item \( O \) — Asymptotic upper bound.
  \item \( o \) — Upper bound that is not asymptotically tight.
  \item \( \Omega \) — Asymptotic lower bound.
  \item \( \neg \) — Logical operator \textit{NOT}.
  \item \( \land \) — Logical operator \textit{AND}.
  \item \( (\ldots, \ldots) \) — A message (tuple) containing several fields.
  \item \( P, E \) — Probability, expectation.
  \item \( Geo(p) \) — Geometrically distributed discrete random variable with success probability \( p \).
  \item \( U(x) \) — Continuous uniform random variable over \([0, x)\).
  \item \( \text{w.h.p.} \) — With high probability. An event occurs w.h.p., if it occurs with probability \( 1 - O(n^{-\alpha}) \) for any fixed \( \alpha \geq 1 \).
  \item \( \text{CAP} \) — Time required for the leader forest to capture all \( n \) nodes with high probability (Lemma 3.3).
  \item \( \text{COV} \) — Time required for a randomized gossip to spread among \( n \) nodes with high probability (Chapter 4).
\end{itemize}
Chapter 1

Introduction

As computing systems continue to evolve from centralized to distributed environments, the criteria for previous design trade-offs between performance and dependability require reconsideration. Faults may occur infrequently for any single component, but they are commonplace for suitably large systems assembled from such components. Some researchers [38] take the position that faults are not rightly understood as a problem to be solved, but rather as a fact of reality to be coped with over time.

In 1974, Dijkstra introduced the notion of *self-stabilization* in the context of distributed systems [16]. He defined a system as self-stabilizing when “regardless of its initial state, it is guaranteed to arrive at a legitimate state in a finite number of steps.” In a system that is not self-stabilizing, an incorrect value in the memory of one of possibly millions participating processors may put the correct system behavior at risk and lead to unpredictable consequences. Self-stabilizing protocols, on the other hand, always behave “nicely”, once the malfunctions cease for a “long enough” time. Motivations for creating self-stabilizing algorithms include overcoming transient hardware malfunctions, resisting misconfiguration, allowing a new (or recovering) processor to join the system without synchronizing it to the global state, and more (see, e.g., [2, 20, 29]). A strong motivation was demonstrated in 1996 by Jayaram and Varghese who showed that crash failures can drive protocols to arbitrary states [42].

Since the introduction of self-stabilization, many such protocols have been devised. In many early self stabilizing protocols every node collects complete, global information, often involving everyone-to-everyone broad-
casts (e.g., [34]). Even in later algorithms, using the local checking and detection paradigm [2, 7, 19, 41], every node still has to observe the state of all its neighbors.

In [13, 14, 36], the authors ask whether it is possible to lower the communication complexity of self-stabilizing protocols below the need of checking every neighbor forever. They define a notion of communication-efficiency for a self-stabilizing algorithm, and derive efficient solutions for many important problems under this metric. Their approach is based mainly on the observation that validating a solution is usually easier than constructing one, and it is thus possible to drastically reduce the communication requirements after the algorithm has stabilized.

In this work, we attempt a different approach, based on using randomized gossip and similar randomized communication processes to spread the information instead of deterministic broadcast, which has been traditionally used in self-stabilizing algorithms. This allows us to reduce the communication complexity of our solutions during the whole execution of the algorithm. We propose efficient solutions for the problems of spanning tree construction, distributed reset and unison. The drawback of this approach is that it is valuable only in graphs that permit efficient dissemination of information by randomized gossip.

A spanning tree of a connected, undirected graph is a tree composed of all its vertices and some (or perhaps all) of the edges. The task of a directed spanning tree construction requires the marking, for each node in a graph, of some of the nodes edges such that the collection of marked edges forms a tree. A spanning tree is a critical component of many other network algorithms, because it allows for efficient routing of messages, dissemination or aggregation of information.

The unison problem is to maintain at every processor a clock showing the same time and running at the same rate [28]. In a synchronous network, running at the same rate is not a problem, since one may count the number of rounds, which are globally synchronized. However, the starting values of the clocks are not necessarily identical. Therefore, one still needs to ensure that all the clocks are synchronized, i.e. their values are equal after every increment.

A reset protocol is an algorithm used to provide to all the nodes a consistent signal [7], usually used to bring all the nodes in the graph to some
predefined state. The reset is initialized by one or more nodes and intuitively involves broadcasting of the request to all the other nodes. More formally, the ideal behavior of a reset protocol can be specified in terms of timeliness, consistency, and causality citevarghese. In a simultaneous reset, such as the one we describe in this work, the transition to the predefined state must occur on the same round for all the nodes.

We present randomized low communication self-stabilizing algorithms for all three of those tasks in a complete, anonymous, synchronous graph. The proposed spanning tree construction algorithm sends a constant number of messages per node every round, and converges within $O(\log n)$ rounds with high probability. The reset and unison algorithms send only one message per node every round, and also converge within $O(\log n)$ rounds with high probability. This results in $O(n \log n)$ messages until convergence w.h.p. for all three algorithms, while all previously known self-stabilizing solutions for those problems have the message complexity of $O(n^2)$ in a complete graph.

Our reset and unison protocols can also be used (although with a different round complexity) in more general classes of synchronous networks. For example, in bounded degree networks their round complexity is $O(D + \log n)$ with high probability, where $D$ is the graph diameter, while still sending one message per node per round.

1.1 Model

A distributed system is a set of cooperating computing elements (processors), interconnected by a network. It is represented by a graph, with processors represented by vertices (also called nodes), and the links between processors represented by graph edges.

In this work, we concentrate on the case of the complete graph topology, where a communication link exists between every pair of nodes. However, the order of outgoing communication links as seen by each processor is arbitrary, possibly predefined by an adversary. It is impossible to determine which link leads to which processor until a message is received from that link directly. In addition, our algorithms do not use any assumption about unique identifiers, that is, our network is anonymous.

We assume that the number of nodes $n$ in the graph is known. This is a reasonable assumption, since we treat the complete network model,
when a processor can know \( n \) simply by counting its outgoing links. The value of \( n \) is used primarily for the computing timing constraints \( COV \) and \( CAP \). Without this assumption (but assuming, e.g., an oracle that returns randomly sampled outgoing links), we would be able to use an upper bound on \( n \) if such was available or could be estimated with sufficiently high probability. Such a bound would then replace \( n \) in the time complexity estimate.

There are two main models of communication for distributed algorithms: \textit{message passing} and \textit{shared memory}. In the message passing model, processors communicate by \textit{sending messages} over the communication links. Each direction of each communication link is represented by a queue of messages sent, but not yet received. In the shared memory model, processors communicate by \textit{reading values in the shared registers} of their direct network neighbors, and writing into their own shared registers. It expresses well a system where shared memory abstraction exists (such as multiprocessor computers) or is implemented on top of the network (e.g., by a special middleware). In this research we treat the message passing model.

We use a synchronous message-passing communication model (as in, e.g., [1]). An execution of a \textit{synchronous} distributed system proceeds in \textit{rounds}. Each round, all the processors make their pending state transitions and possibly send and/or receive messages over their communication links. All the processors are active each round, and all the messages sent during a particular round are received before the beginning of the next round. The links may have infinite capacity, but since during the normal execution every link may contain only \( O(1) \) messages per round, each processor may use finite time processing them and discard the rest. On a side note, [29] shows that no message driven self-stabilizing system can be completely asynchronous.

At any point of time, transient adversarial failures may alter arbitrarily the state of any process variables. However, the adversary is oblivious of, and unable to affect, the random bits generated by the processes. As is common for self-stabilization protocols, when dealing with the algorithms’ time complexity, we always count the number of rounds \textit{after the last failure has occurred}. Therefore, our model of failures is equivalent to the case where the algorithm is initialized to arbitrary values, possibly chosen by an adversary, but no further failures occur when the execution time is measured.
1.2 Our Contribution

Intuitively, one can identify two sources for the high communication complexity. One is the desire to be fast, and hence to communicate with many neighbors at the same time. The other arises from the self stabilization context: even assuming the system is already stabilized, a node must verify that this is indeed the case. For example, in a leader election protocol, every two nodes need to be compared from time to time, to make sure they do not both consider themselves to be the leaders. Electing a leader in a complete network with unique identifiers can be achieved in just one round; however, this would cost $O(n^2)$ messages. Similarly, the time efficient unison algorithms of [6, 9] would stabilize fast when applied to a complete graph, but would have used $O(n^2)$ messages per round.

We present randomized low communication self-stabilizing algorithms for spanning tree construction, distributed reset, and unison. Those are, to our best knowledge, the first self-stabilizing algorithms to use randomized gossip as their main mode of communication. The spanning tree algorithm sends a constant number of messages per node every round, and converges within $O(\log n)$ rounds with high probability. The reset and unison algorithms send one message per node every round, and also converge within $O(\log n)$ rounds with high probability. This results in $O(n \log n)$ messages until convergence w.h.p. for all three algorithms, while all previously known self-stabilizing solutions for those problems have worst case message complexity of $O(n^2)$ in a complete graph.

While the spanning tree algorithm we propose is intended only for the complete graph topology, reset and unison algorithms can function in other graphs with different round complexity (e.g., $O(D+\log n)$ in bounded degree graphs). The results may be meaningful also for a virtual complete graph, e.g. the application layer of the Internet, that is an overlay over a less dense graph. Many modern distributed systems are (virtual) overlay networks, for example, over a large IP-based network. In such a network, a complete graph communication abstraction is provided by the network protocol. Note that in such a virtual complete graph, a node still has (usually) just one (or just a constant number of) actual ports — it cannot send more than a constant number of messages in parallel. Hence, algorithms, such as those given here, that send only a constant number of messages per round should exhibit not
only improved communication complexity but also improved latency, when
compared to algorithms that send messages to all the neighbors.

The “smooth” and uniform behavior of our protocols should also be de-
sirable for the implementation of middleware, since it avoids hard-to-predict
fluctuations in the network load. A middleware protocol that uses unbal-
anced and hard to predict patterns of communications can potentially cause
network congestions and negatively affect not only its own performance, but
also the performance of other applications on the same network.

In Chapter 2, we present a gossip-like random process which is at the
core of this work, and prove that it converges to a spanning tree withing
$O(\log n)$ rounds with high probability. In Chapter 3, we base on it a simple
self-stabilizing algorithm, which succeeds in building a spanning tree if the
node with the highest ID is unique and all the nodes’ clocks are synchronized.

In Chapter 4, we describe an efficient simultaneous reset protocol and
show how to use it to solve the unison problem. In Chapter 5, we use
the protocols from Chapter 4, along with some additional mechanisms to
overcome the limitations of the algorithm from Chapter 3.

Conclusions and directions for future research are presented in Chapter 6.

1.3 Related work

Construction and maintenance of spanning trees in a network is an impor-
tant and well-researched problem. The leader election problem, where all
the nodes must agree on the identity of one of the nodes as the leader, is
closely related. The elected leader may easily impose a spanning tree on the
rest of the graph (see, e.g. [11, 18]). In the other direction, the root of the
spanning tree can be considered the leader. In the context of self stabiliza-
tion, many algorithms were presented for the spanning tree construction in
a general anonymous graph. Those include both the shared memory model,
e.g. [4, 19], and the message passing model [10, 30]; see [27] for a detailed
survey.

The absolute majority of those algorithms either assume unique iden-
tifiers (i.e., are not anonymous), or use the asynchronous model of com-
munication, and therefore are not directly comparable to ours. Moreover,
those that use the shared model of communication usually don’t include ex-
licit communication complexity analysis. All the algorithms we are aware
would use $\Omega(n^2)$ communication in our setting, because in all of them the processors repeatedly read the identifiers of all their neighbors. This is the case even for the leader election protocols from [22], also introduced in the complete graph topology.

The situation is similar with protocols for self-stabilizing distributed reset (e.g., [2, 7, 41]) and unison (e.g., [5, 20, 28]). For example, the reset protocols in [41] either assume the existence of a spanning tree for the propagation of reset messages, and thus would incur at least the cost of spanning tree construction algorithms, or propagate those messages to all the neighbors. In the latter case, $O(n^2)$ messages would be send if $\Omega(n)$ nodes initialized reset requests at once. In the unison algorithms in [5, 20, 28], all the processors read the clock values of all their neighbors repeatedly, so those too would require using either a spanning tree, or $O(n^2)$ communication until convergence when applied in a sufficiently dense graph.

Several definitions of communication efficiency for self-stabilizing algorithms have been proposed. [13] deals with self-stabilizing leader election when process identifiers exist. It defines a leader election algorithm to be communication-efficient if it eventually uses only $n-1$ unidirectional links, which is optimal for that problem. [14] and [36] introduce the notions of $\diamondsuit$-$k$-stability and $\diamondsuit$-$k$-communication efficiency for the shared memory model. An algorithm is $\diamondsuit$-$k$-stable if, in every its possible execution, eventually each process reads communication variables of at most $k$ different neighbors. An algorithm is $\diamondsuit$-$k$-communication efficient if eventually all the processes together use at most $k$ incoming communication links. In both cases the $k$ is taken not over one round, but as a union of communication links used in all the rounds of the relevant execution suffix. [36] investigates the possibility of communication-efficient spanning tree construction with process identifiers, defining communication efficiency as $\diamondsuit$-$O(n)$-communication efficiency. [14] shows that for many non-trivial problems, such as coloring, maximal matching, and maximal independent set, it is impossible to get self-stabilizing solutions where every participant communicates with less than every neighbor in the shared memory model. On the other hand, they present protocols for those 3 problems where a fraction of the processors communicates with exactly one neighbor in the stabilized phase.

[13, 14, 36] define an algorithm to be communication-efficient based on the number of different neighbors with which each node communicates dur-
ing the whole stabilized phase. Because of the gossip-like nature of our algorithms, every node communicates with many neighbors, but it does so only a few neighbors at a time. Also, those papers concentrate on reducing communication after the stabilization is complete. We, on the other hand, are interested in developing protocols that are message-efficient at any time, and not only after the stabilization. Therefore, we use the more traditional notion of communication efficiency, counting the number of messages sent before the stabilization, and during every round after the stabilization.

Many of the techniques we use in our algorithm have been presented before. In our spanning tree algorithm, we limit the branching degree to reduce worst-case communication. Moreover, in the reset and unison algorithms, a node is allowed to send message only to one neighbor per round. The spanning tree protocol presented in [30] also uses a bounded degree spanning tree to reduce communication and memory requirements, however, the authors provide no performance analysis for their algorithm. The idea of contacting only one neighbor at a time was used for reducing memory requirements in various papers, such as in [8]. For deterministic algorithms, this approach often increases the stabilization time.

The technique of repeatedly recoloring a tree with random colors to detect competing trees was presented in [2, 4], and several other publications. We use it with a slight modification in Chapters 5.1-5.2, namely, in our algorithms only the roots participate in detection.

Afek and Matias used randomization in [3] to solve non-self-stabilizing leader election in a complete network with $O(n)$ messages. They reduced communication by distributively selecting a sparse random subgraph, which with high probability will be connected, and then applying a special random identifier selection mechanism. This mechanism was adapted to the self stabilizing context in [4] to achieve improved time complexity, and we use it in our spanning tree algorithm for the same purpose.

Our work is closely related to randomized gossip and random branching processes [15, 24, 32, 35, 43]. The spanning tree algorithm we propose is based on a random branching process, and the reset and unison algorithms use randomized gossip as underlying communication mechanism. Gossip-based protocols are popular because they are simple and often naturally self-stabilizing, but the tasks that can be solved by them are usually rather limited. We hope that this paper can help to bridge the gap and combine
the simplicity and robustness of randomized gossip with the more structured approach.

The growth of the random branching process on which our spanning tree is based can be described by a recurrence similar to that of the “infect forever” process analyzed by Pittel [39]. However, to our best knowledge, the exact process we describe has never been presented before. [39] is almost universally cited to support the claim that randomized gossip requires $O(\log n)$ rounds with high probability. However, its focus is quite different, dealing with refinement of convergence limit. Based on the theorems presented there, one would only be able to say that as $n \to \infty$, the probability that the number of rounds required is $\log_d n + \frac{\log n}{d-1} + O(1)$ (where $d$ is a configurable constant defined in the next section) converges to 1.

The proof of convergence with high probability, limited to the case when one message is sent per process (equivalent to $d = 2$), is found in [26], and bounds the deviations of the process behavior from that limit. That proof is very elegant, but rather involved mathematically, and is presented on 7 pages, not including one of the lemmas from the same paper that it uses. We have developed our own proof of a weaker claim, which is sufficient for our goals, and which we present in Chapter 2. We believe our proof may be easier to understand for researchers with standard computer science background.
Chapter 2

The random branching process

In this section, we describe a simple random process that serves as a starting point for our construction. It grows a spanning BFS tree with degree (maximum branching factor) $d \geq 2$ in a complete graph with $n$ nodes, starting from a given root node. $d$ is a configurable constant that can be chosen by the user based on, e.g., bandwidth and memory limitations.

The execution alternates between request and confirmation rounds; the order of those rounds is known to all the nodes. The tree starts with exactly one node, the predefined root of the tree, all the other nodes are available for capture. Every node in the tree can have at most $d$ children. Every request round, every node $v$ in the tree that has $d(v) < d$ children from the previous rounds, selects $d - d(v)$ additional nodes as its children uniformly and independently at random\(^1\). We use the term selections to denote the union of all the randomly drawn nodes, with repetitions.

After the request round is complete, all the nodes that were selected this way join the tree during the confirmation round. A newly joining node chooses one of the tree nodes that selected it arbitrarily and notifies its new parent. When a node which is already in the tree (i.e. already has a parent) is selected, it may remain with the old parent, or pick one of the new ones. We shall see that it does not matter for our analysis.

\(^1\)We assume that a node selects new children independently, and can even select itself to simplify the analysis. In real-world applications, it is not reasonable to select the same node more than once, or to select nodes which are already in its child/parent set.
Let $T_i$ be the number of nodes in the tree after $i$ request and $i$ confirmation rounds have completed, starting with $T_0 = 1$.

The number of new random messages (selections) sent by a particular node depends on the number of children it currently has. This number has a complex probability distribution which is potentially a function of the whole tree structure.

On the other hand, the total number of selections is always $T_i(d - (T_i - 1)) = T_i(d - 1) + 1$, because the maximum degree is $d$, and there are exactly $T_i - 1$ existing tree edges, regardless of tree structure. An equivalent explanation to this fact is that this is also the number of leaves in a $d$-regular tree with $T_i$ internal nodes. This number is linear in the number of tree nodes: $T_i(d - 1) + 1 > T_i(d - 1)$. The number of joining nodes is non-decreasing in the number of selections, i.e., by drawing additional selected nodes we cannot slow down the growth of the tree. Therefore, we may bound the number of tree nodes from below by a randomized gossip model, where every tree member sends a message uniformly at random to $d - 1$ nodes each round.

**Theorem 2.1.** The random process described in this chapter creates a spanning tree of the graph within $O(\log n)$ rounds on average and with high probability.

It is well known (e.g., [32]), that even for $d = 2$, messages from the root will reach all the nodes with high probability after only $O(\log n)$ rounds. Frieze and Grimmett show that, for random process equivalent to having $d = 2$, the number of rounds $S_n$ has the following property:

**Theorem 2.2** (Adapted Theorem 5.2 from [26]). If $\gamma > 0$ is constant, then, for all $\varepsilon > 0$, $P(S_n > (1 + \varepsilon) \alpha(\gamma) \log 2 n) = o(n^{-\gamma})$, where $\alpha(\gamma) = 1 + (\gamma + 1) \log 2$, is a tight multiplicative constant.

However, the formal proof was difficult to find at first, as almost none of the papers (except [39]) seem to cite it. Without seeing formal proof, we could not easily generalize the claim to more types of processes (see Lemma 2.10 toward the end of this section). During this research we developed our own, alternative, proof of Theorem 2.1, which we present in the next section. We believe that our proof uses simpler techniques and may be of independent interest.
2.1 Proof of Theorem 2.1

We define the random series: \( G_i = \frac{T_i}{n} \). Since the random choices are made independently and uniformly at random among all the nodes, this process can be analyzed using the standard balls-in-bins model (see, for example, [37]). The expected number of nodes not selected by any node in the tree is:

\[
\begin{align*}
n \left(1 - \frac{1}{n}\right)^{T_i(d-1)+1} &= n \left(1 - \frac{1}{n}\right)^{G_i(n(d-1)+1)} \\
&\leq ne^{-G_i(d-1)} \quad (2.1)
\end{align*}
\]

The expected number of nodes selected by at least one tree node at least:

\[
\begin{align*}
n - ne^{-G_i(d-1)} &= n(1 - e^{-G_i(d-1)}) \\
&= n(1 - e^{-G_i(d-1)}) \quad (2.2)
\end{align*}
\]

The expected proportion of the selected nodes which were not in the tree before this round is \(1 - T_i/n = 1 - G_i\), because the event that a node is in a tree and the event that it was selected are independent. We conclude that the average number of new nodes joining the tree during confirmation round \(i\) is at least \(n(1 - G_i)(1 - e^{-G_i(d-1)})\). This results in the following recursive relation:

\[
\begin{align*}
\mathbb{E}(G_{i+1}|G_i) &\geq G_i + (1 - G_i)(1 - e^{-G_i(d-1)}) > 1 - (1 - G_i)e^{-G_i(d-1)} \quad (2.3)
\end{align*}
\]

Our aim is to prove that the spanning tree grows exponentially fast before it absorbs all the nodes. Intuitively, it grows fast during the first rounds because there are a lot of nodes outside the tree, and the chance for several tree nodes to select the same outside node is small. During the last rounds, it grows fast because the tree has a lot of leaves, and the chance that an outside node will not be selected is small. We start with proving that the claim holds on average using some basic calculus, and later expand the proofs with a few probability concentration bounds to show this with high probability.

2.1.1 Expected convergence speed

Our first lemma is that on average, the tree absorbs a constant proportion of the nodes after only a logarithmic number of rounds.
Lemma 2.3. For any constant \( c \in (0,1) \), there exists \( t_1 \in O(\log n) \) such that \( \mathbb{E}(G_{t_1}) \geq c \).

Proof. Define the growth rate \( R_i = G_{i+1}/G_i \). We aim to prove that \( \mathbb{E}(R_i | G_i < c) \geq 1 + \varepsilon(c) > 1 \).

\[
\mathbb{E}(R_i | G_i) = \frac{\mathbb{E}(G_{i+1} | G_i)}{G_i} > 1 - (1 - G_i)e^{-G_i(d-1)} = G_i^{-1} + (1 - G_i^{-1})e^{-G_i(d-1)} \tag{2.4}
\]

Letting \( x = G_i^{-1} \), define \( r(x) \) as:

\[
r(x) \equiv x + (1 - x)e^{-\frac{d-1}{x}} < \mathbb{E}(R_i | G_i = 1/x) \tag{2.5}
\]

If \( G_i < c \), we know that \( \frac{1}{c} < x \leq n \). We will prove that \( r(x) \) is strictly increasing in this range, and thus can be bounded from below by \( r(\frac{1}{c}) \). To this end, we compute its derivatives:

\[
r'(x) = 1 - e^{-\frac{d-1}{x}} \left( 1 + \frac{(d-1)(x-1)}{x^2} \right) \tag{2.6}
\]

\[
r''(x) = e^{-\frac{d-1}{x}} \left( d - 1 - x(d+1) \right) \frac{d-1 - x(d+1)}{x^4} \tag{2.7}
\]

Note that \( \lim_{x \to \infty} r'(x) = 0 \) and \( r'(x) \) is continuous for \( x > 0 \). The only extreme point of \( r'(x) \) is at \( x = \frac{d-1}{d+1} < 1 \) and \( r'(1) = 1 - e^{-d+1} > 0 \), so \( r'(x) \) must approach 0 from above, i.e. \( r'(x) \geq 0 \) for all \( x \in (1, \infty) \).

This means that \( r(x) \) is nondecreasing for \( x \in (1, \infty) \). Since \( x > \frac{1}{c} > 1 \), \( R_i > r(x) > r(\frac{1}{c}) \). It remains to show that \( r(\frac{1}{c}) > 1 \). Observe that

\[
r(\frac{1}{c}) = \frac{1}{c} + \left( 1 - \frac{1}{c} \right) e^{-c(d-1)} > 1
\]

can be reduced by subtracting \( e^{-c(d-1)} \) from both sides to

\[
\frac{1 - e^{-c(d-1)}}{c} > 1 - e^{-c(d-1)} \tag{2.8}
\]

which is always true for \( c \in (0,1) \). Recall that \( G_0 = 1/n \) and we are looking for \( t_1 \) such that \( G_{t_1} \geq c \). Since \( G_i \) increases in the expectation by more than
a multiplicative factor of \( r(\frac{1}{c}) \), \( t_1 \) is strictly less than

\[
\log_r(\frac{1}{c}) \cdot \frac{c}{c n} = \log_r(\frac{1}{c}) \cdot \frac{\log n + \log c}{\log r(\frac{1}{c})} = \frac{O(\log n) + O(1)}{O(1)} = O(\log n)
\]

\[
(2.9)
\]

Now define \( F_i = 1 - G_i \). By applying some algebra to (2.3), we get:

\[
\mathbb{E}(F_{i+1}|F_i) < F_i e^{-(1-F_i)(d-1)}
\]

(2.10)

Intuitively, \( F_i \) is the fraction of nodes still not absorbed after \( i \) phases. Our second lemma is, that once the tree has absorbed a constant proportion of the nodes, it takes at most a logarithmic number of rounds to absorb the rest.

**Lemma 2.4.** For any constant \( c \in (0, 1) \), there exists \( t_2 \in O(\log n) \) such that \( \mathbb{E}(F_{t_1+t_2}|F_{t_1} \leq 1-c) \leq \frac{1}{2n} \).

**Proof.** Define the extermination rate \( \tilde{R}_i = F_{i+1}/F_i \). This time we will prove that \( \mathbb{E}(\tilde{R}_i|F_i \leq 1-c) \leq \varepsilon(c) < 1 \).

\[
\mathbb{E}(\tilde{R}_i|F_i \leq 1-c) < e^{-(1-F_i)(d-1)} \leq e^{-c(d-1)} < 1
\]

(2.11)

Let \( \varepsilon(c) = e^{-c(d-1)} < 1 \) as above\(^2\). The expected number of request-confirmation round pairs is at most

\[
\log_c \left( \frac{F_{t_1}}{F_{t_1+t_2}} \right) = \log_c \frac{2n(1-c)}{c(d-1)} = \frac{\log 2 + \log(1-c) + \log n}{c(d-1)}
\]

(2.12)

This expression is \( O(\log n) \).

Recall that every node sends messages to at most \( d+1 \) other nodes during every 2 rounds (\( d \) children and possibly the chosen parent). We have the following corollary:

\(^2\)A particular node outside the tree is not assimilated if none of the tree’s \( T_i(d-1) + 1 \) independent choices hit it. Since \( T_i/n \geq c \), the probability of this event is less than \( e^{-c(d-1)} \). This is also the expectation on the average proportion of nodes not assimilated (using the balls in bins model).
Corollary 2.5. The random process described in this chapter creates a spanning tree of the graph within $O(\log n)$ rounds on average.

2.1.2 Probability bounds for the convergence speed

Lemma 2.6. For any $\alpha \geq 1$, we have that $\mathbb{P}(G_{t_1} \geq \frac{1}{2}) \geq 1 - O(n^{-\alpha})$ for $t_1 = 252\alpha \log \frac{n}{2}$.

Proof. Let us return to the proof of Lemma 2.3. We continue using the same notation. Recall that $R_i = G_{i+1}/G_i$. First, we would like to bound from below the probability $\mathbb{P}(R_i \geq \frac{5}{4} \mid G_i < \frac{1}{2})$.

By substituting $x = 2$ into (2.5), and using the fact that $d \geq 2$ and the lower bound on $\mathbb{E}(R_i \mid G_i < \frac{1}{2})$ is non-increasing in $G_i$, we get $\mathbb{E}(R_i \mid G_i < \frac{1}{2}) > 2 - e^{-\frac{1}{2}} > \frac{5}{4}$.

Define new random series $X_i = d + 1 - R_i$. Since $R_i \in [0, d+1]$, all the $X_i$ are non-negative, and we can use the Markov inequality on the m:

$$\mathbb{P}(R_i \geq \frac{5}{4} \mid G_i < \frac{1}{2}) = 1 - \mathbb{P}(R_i < \frac{5}{4} \mid G_i < \frac{1}{2})$$

$$= 1 - \mathbb{P}(X_i > d - \frac{1}{4} \mid G_i < \frac{1}{2}) \geq 1 - \frac{\mathbb{E}(X_i \mid G_i < \frac{1}{2})}{d - \frac{1}{4}} > \frac{\frac{3}{4} - e^{-\frac{1}{2}}}{d - \frac{1}{4}} > \frac{1}{7d} \tag{2.13}$$

Note, that this holds for each $R_i$ independently, since $R_i$ conditioned on a particular $G_i$ depends only on the random choices made during request round $i$, and not, e.g., on any other $R_{j}$ for $j < i$.

Consider the dependence on $d$ in (2.13). For particular $G_i$ and $d$, the complete distribution of $R_i$ depends on the $T_i(d-1)+1$ independent random choices made by the nodes. For $d' = d+1$ and the same distribution of the first $T_i(d-1)+1$ choices, the same nodes are captured, and possibly more can be acquired by the additional $T_i > 0$ choices. Therefore, $R_i$ for $d' = d+1$ must dominate the one for $d$, because additional messages cannot prevent capture of nodes. The bound (2.13) holds for $d = 2$, and the same numerical

3We chose $\frac{5}{4}$ as an arbitrary number between 1 and $\mathbb{E}(R_i \mid G_i < \frac{1}{2})$. It is likely that the constants we use here can be optimized.

4A random variable $A$ is stochastically dominant over a random variable $B$ if $A \geq B$ in every possible state of nature.
bound must hold for any distributions of $R_i$ that dominate it for $d > 2$ (since they must lead to a convergence at least as fast):

$$P \left( R_i \geq \frac{5}{4} \mid G_i < \frac{1}{2} \right) > \frac{1}{14} \quad (2.14)$$

To prove the Lemma we need to bound the probability

$$P \left( G_{t_1} < \frac{1}{2} \right) = P \left( G_0 \prod_{i=1}^{t_1} R_i < \frac{1}{2} \right) = P \left( \prod_{i=1}^{t_1} R_i < \frac{n}{2} \right) \quad (2.15)$$

Consider 2 cases: if for any $i < t_1$ it holds that $G_i \geq \frac{1}{2}$, then $G_{t_1} \geq \frac{1}{2}$ because $G_i$ is non-decreasing. Otherwise, as we have mentioned earlier, bound (2.14) holds for every $R_i$ independently. Since $R_i \geq 1$, it is sufficient to prove that

$$P \left( \{ i : R_i > \frac{5}{4} \} \mid \log \frac{n}{2} \right) < P \left( \{ i : R_i > \frac{5}{4} \} \mid \frac{9}{2} \log \frac{n}{2} \right) \quad (2.16)$$

is $O(n^{-\alpha})$.

We use the following bound on the lower tail of a Binomial distribution, which can be derived by a straightforward application of Hoeffding’s inequality [31]: if $S$ is $Bin(m, p)$, then for any $p' < p$

$$P \left( S \leq mp' \right) \leq e^{-2m(p-p')^2} \quad (2.17)$$

In our case, we have $m = t_1$, $p = P \left( R_i \geq \frac{5}{4} \mid G_i < \frac{1}{2} \right) > \frac{1}{14}$. Substituting $p' = \frac{9}{2} \log \frac{n}{2} = \frac{1}{56\alpha}$,

$$P \left( \{ i : R_i > \frac{5}{4} \} \leq \frac{9}{2} \log \frac{n}{2} \right) \leq \exp \left( -2t_1 \left( \frac{1}{14} - \frac{1}{56\alpha} \right)^2 \right)\leq \exp \left( -2t_1 \left( \frac{3}{56} \right)^2 \right) = O(n^{-\alpha}) \quad (2.18)$$

To prove that the bound in Lemma 2.4 also holds with high probability, we turn to the general framework for analyzing stochastic recurrences
introduced by Karp [33].

Suppose we have a randomized algorithm that on input of size \( x \), performs “work” \( a(x) \), and then produces a subproblem of a random size \( H(x) \), which is then solved by recursion. Assume that there is a constant \( d \) such that \( a(x) = 0 \) for all \( x < d \), i.e., when the input becomes small enough the recursion stops, and that \( H(x) \) is a random variable depending only on \( x \). The algorithm’s time complexity on input \( x \) can then be written down as a recurrence

\[
T(x) = a(x) + T(H(x)).
\]

The following is a simplified version of Theorem 1.1 from [33] adopted from [23]:

**Theorem 2.7** (Karp’s Theorem). Let \( T(x), H(x), \) and \( a(x) \) be as above. Assume that \( E(H(x)) \leq m(x) \) for some deterministic function \( m(x) \), such that \( m(x) \in [0, x] \) for all \( x \). Let \( u(x) \) be the (deterministic) solution for the recurrence \( u(x) = a(x) + u(m(x)) \). If \( a(x) \), \( m(x) \), and \( \frac{m(x)}{x} \) are all non-decreasing, then for any \( w > 0 \):

\[
P(T(x) > u(x) + wa(x)) \leq \left( \frac{m(x)}{x} \right)^w
\]

**Lemma 2.8.** For any \( \alpha \geq 1 \), \( t_1 \) such that \( F_{t_1} \leq 1/2 \) as in Lemma 2.4, and \( t_2 = \frac{2(\alpha+1)}{d-1} \log n \) it holds that \( P(F_{t_1+t_2} \leq \frac{1}{2n}) \geq 1 - n^{-\alpha} \).

**Proof.** We identify the size \( x \) of the problem as the proportion of non-tree nodes \( F_t \), i.e. \( H(F_{t_1}) = F_{t_1+1}, H(F_{t_1+1}) = F_{t_1+2}, \) etc. In our case, \( a(x) = 1 \) and from (2.10) we get \( m(x) = xe^{-(1-x)(d-1)} \geq E(H(x)) \). Also, \( u(\frac{1}{2}) \leq \frac{2}{d-1} \log n \) by substituting \( c = 1/2 \) into (2.12). The reader can easily verify that \( 0 \leq m(x) \leq x \), and that \( m(x) \) and \( \frac{m(x)}{x} \) are nondecreasing functions. Applying Theorem 2.7:

\[
P \left( T - t_1 \geq \frac{2}{d-1} \log n + w \right) \leq e^{-\frac{d-1}{2}w}
\]

Substitute \( w = \frac{2\alpha}{d-1} \log n \) to get the desired result. \( \square \)

**Corollary 2.9.** The random process described in this chapter creates a spanning tree of the graph within \( O(\log n) \) rounds with high probability.

Although the constants that appear in the above lemmas are quite large, our simulations show that in reality much less time is required in most cases.
In fact, [26] shows that the multiplicative factor should be at most $\alpha + 1 + \frac{1}{\log 2}$ even when $d = 2$.

## 2.2 A generalization

Having proved Theorem 2.1, we develop a tool that would allow us to get similar results for a more general class of algorithms through reduction.

Let $A$ be a synchronous algorithm on $n$ nodes, with a state that can be defined in terms of the number of nodes in a particular structure. Define $G^A_i$ for this algorithm in the same way we have defined $G_i$. The following lemma will be essential for the next chapters:

**Lemma 2.10.** Assume that $G^A_{i+1} \in [G^A_i, (d + 1)G^A_i]$. If $E(G^A_{i+1}|G^A_i = x) \geq E(G_{i+1}|G_i = x)$, then $A$ converges within $O(\log n)$ rounds with high probability.

**Proof.** Lemma 2.3 applies to $A$ because

$$1 < 1 + \varepsilon(c) < E\left(\frac{G_{i+1}}{G_i} \bigg| G_i = x \leq c\right) \leq E\left(\frac{G^A_{i+1}}{G^A_i} \bigg| G^A_i = x \leq c\right) \quad (2.20)$$

I.e. since the expected growth of $G_i$ is bounded from below by a constant multiplicative factor greater than one, and $G^A_i$ grows round-by-round at least as fast, its growth must be exponential as well.

Let $F^A_i = 1 - G^A_i$; therefore $E(F^A_{i+1}|F^A_i = x) \leq E(F_{i+1}|F_i = x)$. Lemma 2.4 now applies for the same reasons.

We based our proof of Lemmas 2.6 and 2.8, only on the expected rate of growth and bounds that hold for $A$ as well, therefore the convergence must be exponentially fast with high probability. 

**□**
Chapter 3

Spanning forest algorithm

First, we show a simple self-stabilizing extension of the protocol in Chapter 2. This is not the final algorithm — we present it mainly to prove some basic lemmas that are used later on. The algorithm constructs a spanning forest, which is a relaxation, in a way, of the concept of a spanning tree. A spanning forest is any subgraph that is both a forest (contains no cycles) and spanning (includes every vertex in the graph). This spanning forest becomes a spanning tree under two assumptions:

- There is a single node with the highest identifier.
- All the nodes are provided with a globally synchronized clock signal.

Those assumptions will be removed in Chapter 5 by providing additional mechanisms that provide them with high probability without asymptotically increasing the algorithm’s time and message complexity.

Like many other spanning tree algorithms that are based on local checking and detection (e.g., [2, 19]), for maintaining a tree, we provide every node with two local variables: treeid and treedistance. All the error-free nodes in a single tree share the same treeid, and all, except for the roots, have a positive treedistance which is larger by one than that of their parent. Nodes with treedistance equal to 0 consider themselves roots. The algorithm ensures that all the nodes eventually join the trees with the largest treeid in the graph. The pseudocode performed by every node is the same and appears on Figures 1–3.

When one node captures another node that already has children, we can use this fact to capture the subtree of the new node as fast as possible.
The new node’s “effort” from the earlier rounds is preserved to some degree. Intuitively, it is more likely to receive confirmation from its old children than from randomly picked nodes, because we know its children did not join the tree yet during the previous confirmation round. Therefore, a newly joined node does not start with an empty children set, like in the process in Chapter 2. We were unable to account for this optimization in our analysis (in fact, we do not use treedistance in any of the convergence speed proofs in Chapter 3.1), but our simulations suggest that it has a measurable (although not asymptotic) impact on the convergence speed (see Figure 8).

To be able to capture subtrees quickly, we would also like the trees to be of a small height. A node prefers, among all potential parents with the same treeid, the ones with the smallest treedistance. The trees “contract”: new nodes join as close to the root as possible, and also move closer to the root within the same tree whenever offered an opportunity. If allowed enough time, this process would converge to an (almost) full tree of degree $d$, because all the levels close to the root would fill up eventually.

For the simplicity, we define the order on $<\text{treeid}, \text{treedistance}>$ pairs to be increasing by treeid, and then decreasing by treedistance. Any comparisons in the pseudocode use this order.

**Figure 1** Spanning forest algorithm, main loop

1: loop
2:   $R \leftarrow$ received request messages
3:   $C \leftarrow$ received confirmation messages
4:   if request then // this is the request round
5:       run request procedure (Figure 2)
6:   else
7:       run confirmation procedure (Figure 3)
8:  end if
9:  request $\leftarrow \neg$request
10: end loop

If the current round is a request round for the node, it collects all the confirmation messages sent by its children on the previous round and updates its children variable. The nodes that sent no confirmation are no longer its children. It selects additional children as required and sends both its current and newly selected children its $<\text{treeid}, \text{treedistance}>$ pair
in a request message.

**Figure 2** Spanning forest algorithm, request procedure

1: $children \leftarrow C$
2: **while** $|children| < d$ **do**
3: $children \leftarrow children \cup$ random neighbor
4: **end while**
5: send request $\langle treeid, treedepth \rangle$ to the first $d$ children

During a confirmation round, the node checks whether it has received any request messages from nodes with identifier larger than its current parent. If such request messages were received, it must accept the best potential parent and send it a confirmation message. A node does not preserve its parent’s identity between rounds, but it can deduce its $treeid$ and $treedepth$ from its own. When implementing the algorithm, it may be beneficial to prefer the original parent over any other equally good requests, to avoid excessive parent switching. For simplicity, here we describe an algorithm that chooses an arbitrary parent among those with the largest identifiers.

The node collects all the received $\langle treeid, treedistance \rangle$ pairs that can potentially become its parent, and looks for the messages with the largest $treeid$. Among those, it chooses one of those with the smallest $treedistance$. The source of the winning message becomes this node’s parent. If no suitable parent has sent a request during its confirmation round, a node becomes a root.

**Figure 3** Spanning forest algorithm, confirmation procedure

1: $candidates \leftarrow \{ r \in R : r > \langle treeid, treedepth \rangle \}$
2: **if** $|candidates| > 0$ **then**
3: $parent \leftarrow \max (candidates)$
4: $treeid \leftarrow parent.treeid$
5: $treedepth \leftarrow parent.treedepth + 1$
6: send confirmation $\langle \rangle$ to $parent$
7: **else** // we are orphaned, become a root
8: $treedepth \leftarrow 0$
9: **end if**

Note that the algorithm of Chapter 2 alternates between request and confirmation rounds. In the context of self-stabilization, a round may be a
request round at one node and a confirmation round at another. We say that two nodes are synchronized if their request rounds coincide. Handling the lack of synchronization directly in the spanning forest algorithm would make it and the related proofs much less elegant. We ignore this problem for now, by not claiming anything about the algorithm’s properties when the configuration is not perfectly synchronized; it will be solved in Chapter 5.

### 3.1 Algorithm analysis

For this section we define the true parent of a non-root node \( v \) to be the node to which \( v \) sent the confirmation message during its last confirmation round, but only if it has the same treeid as \( v \). Otherwise \( v \) has no current true parent. Note that there can be several scenarios when a node does not have a true parent. The obvious ones are when \( v \) is a root, or if it or its parent have been corrupted. Another possibility is when \( v \)'s parent joins a different tree during the last confirmation round.

**Lemma 3.1.** Assume that all the nodes are synchronized. At every point of time after the first 3 rounds, a node \( u \) is the true parent of a node \( v \) only if \( v \) is a child of \( u \). The graph formed by the (directed) true parent relationships is a directed forest with a branching factor at most \( d \).

**Proof.** After the first round, node \( v \) cannot send a confirmation message to \( u \), unless it receives a request message from \( u \). During the first 3 rounds, \( v \) executed the confirmation procedure at least once after a fail-free round, therefore, \( u \) has indeed sent a request message to \( v \), so \( v \) must have been in \( u \)'s children set at that time. The only scenario when \( u \) can lose \( v \) as a child, is if \( v \) does not respond to one of \( u \)'s later request messages. But this means that \( v \) has responded to some other node, and \( u \) was not the last node to which it sent a confirmation message, in contradiction.

The relationship is cycle-free, because a node only considers parents whose \(<\text{treeid},\text{treedistance}>\) pair is lexicographically larger than its own, which is an asymmetric and transitive relationship. After 3 rounds, we can be sure that every node has based its parent choice on a real non-corrupted request message. A node’s \(<\text{treeid},\text{treedistance}>\) pair value cannot decrease\(^1\), and cannot increase beyond the value of its parent, so

---

\(^1\)After Chapter 5, this claim will be correct only between resets and after the last reset

---
this relationship is preserved over time.

Furthermore, every node has at most one true parent, and the branching degree of every node is less than or equal to \( d \), because every node can have at most \( d \) children at any given time. \( \square \)

**Lemma 3.2.** Assume that there is only one node with the maximum `treeid` in the graph, and all the nodes are synchronized. Then, when running the spanning forest algorithm, that node’s tree absorbs all the nodes in the graph within \( O(\log n) \) rounds with high probability.

**Proof.** We base our proof on Lemma 2.10, comparing our algorithm to the process in Chapter 2. The choices to which new nodes to send request messages are made independently and uniformly at random. Therefore both processes can be analyzed using the standard balls-in-bins model (see, for example, [37]). Such a model involves two independent factors: the number of balls thrown (messages sent), and the probability per ball of hitting a bin of a particular type (a node which is not already in the tree).

In order to show that \( E(T_{i+1} | T_i = x) \geq E(T_{i+1} | T_i = x) \), it suffices to prove that when the tree contains \( T_i \) nodes: (1) at least \( T_i(d-1) \) selections are made; and (2) every such selection is independent and hits a node in the tree with probability at most \( T_i/n \). The first claim holds for the same reason as in Chapter 2: recall that this is the number of leaves in a \( d \)-regular tree with \( T_i \) internal nodes.

The second item would be correct if all the selections would be new random selections (line 3 in the request procedure), simply because the probability of randomly hitting a tree node is \( T_i/n \). However, as we have mentioned earlier, a new node that already had children before joining the tree may preserve them. This did not happen in the original process.

We assume that all the nodes are synchronized, therefore the next time those children receive any request messages is together with the request messages from their original parent. Assume that one such child node receives \( k \) such messages. Only one message, the one from its original parent, was sent to it deterministically. The other \( k-1 \) messages must be new random selections, made uniformly and independently from each other.

Now consider the event that \( k \) new random selections messages arrive at some node. For \( k \) independent uniformly distributed random variables event.
U_1, \ldots, U_k, it holds that \( P(U_2 = U_1, \ldots, U_k = U_1) = P(U_2 = x, \ldots, U_k = x) \)
for any \( x \) s.t. \( P(U_1 = x) \neq 0 \). Therefore, the probability of this event is the same as the probability of the event we described in the previous paragraph. The difference is, in the former, the node to which the messages were sent was already in the tree with probability \( 0^2 \), and in the latter with a probability of \( \frac{T_i}{n} \). We conclude, therefore, that in the former case the expected number of captured nodes is at least as large as if all the selections were new independent random selections.

Finally, \( T_{i+1}^A \in [T_i^A, (d + 1)T_i^A] \) for the same reasons as for the process in Chapter 2.

We define the leader forest to be the forest of nodes with the largest \textit{treeid} value in the graph (it is a forest because of Lemma 3.1). Once again, we do not claim anything about the speed of convergence when the nodes are not synchronized, since our final spanning tree algorithm will make sure eventually this is the case.

**Lemma 3.3.** Assume that all the nodes are synchronized. When running the spanning forest algorithm, the leader forest absorbs all the nodes in the graph within \( O(\log n) \) rounds with high probability.

**Proof.** It is not hard to see, that eventually all the nodes in the graph must be captured by one of the trees in the leader forest. We compare its growth with the growth of a single dominating tree from Lemma 3.2.

Let \( q \) be the number of different roots in the leader forest, and \( T_{i,j}^A \) be the number of nodes in the tree of root \( j \) after \( i \) confirmation rounds. While the single tree in Lemma 3.2 performs exactly \( T_i(d - 1) + 1 \) selections during the following request round, the leader forest with the same total number of nodes tries at least \( \sum_j \left( T_{i,j}^A(d - 1) + 1 \right) = T_i^A(d - 1) + q \). The probability of hitting a node outside the leader forest is the same in both cases, as it depends only on the local tree structure and the total number of nodes.

As the leader forest makes at least as many selections with at least the same probability of success as in Lemma 3.2, we can apply Lemma 2.10 to it as well.

---

\(^2\)Such a child cannot be in the tree when it sends the confirmation to the parent, otherwise they would not send confirmations for their parent’s previous, inferior, \textit{treeid}.

\(^3\)Recall that there may be more than one tree with the largest \textit{treeid}.

26
As the reader can see, the algorithm of this section is not very useful, unless we can ensure that there is only one root with the highest `treedid` and synchronize all the nodes to the same round. We show how to do it in the following sections.
Chapter 4

Low-bandwidth self-stabilizing reset

In the ordinary randomized gossip [26], every node with a “rumor” sends it to a single random neighbor every round. Let COV be the number of rounds after which such a rumor becomes known to all the nodes with high probability. It is well known that in a complete graph COV is $O(\log n)$ (the relevant random process is described by the right part of equation (2.3) when $d = 2$).

We add a variable $\text{reset}$, to the node state. Any node can start a reset procedure by setting its $\text{reset}$ to $2\text{COV}$. The only exception are the nodes that already have a non-zero $\text{reset}$ variable; those are the nodes that already participate in a reset. The $\text{reset}$ value is used as a countdown “time-to-live” timer: every round when the timer is still positive, the node decreases it by 1 and sends its value to a random neighbor. When the value goes from 1 to 0, the node resets. The intuition behind using $2\text{COV}$ is that it takes COV rounds for all the nodes to become aware that a reset is in progress and another COV rounds to agree on when to perform the reset.

When a node receives a $\text{reset}$ timer value from another node, it takes the maximum between all the received values and its own as the new $\text{reset}$ value. However, the nodes never set their timer to values larger than $2\text{COV}$, since those were clearly introduced by an adversary.

**Theorem 4.1.** Let $i$ be a round when one of the nodes initializes a reset procedure. Then, with high probability, during some round $t$, such that $i +
2COV ≤ t < i + 3COV, all the nodes in the graph reset simultaneously (and on round t + 1 all the reset variables are 0).

Proof. Let v be a node that initialized a reset on round i, and let the current round be j. As v counts down its timer, there could be new resets initialized by nodes other than v; there can also be older resets that were initialized before round i.

Consider the way messages from v spread among the nodes in the graph. Whenever a message reaches a node u that does not participate in a reset, or participates in a reset that started before round i, that node has a smaller reset. In this case, u adopts the new timer value. Whenever a message reaches a node u with the same, or a higher reset value (i.e. u participates in a reset that was initialized during round i or later), u ignores it. In both cases, u goes on to spread messages with reset ≥ 2COV − (j − i).

Because every value spreads to all the nodes w.h.p. within COV rounds, during round i + COV, all the nodes have reset ≥ COV with high probability. Therefore, no new resets are initialized during rounds i + COV...i + 2COV, and the maximal value of reset remains consistent during those rounds (max(reset) + j is constant).

Applying the same theorem again, tells us that, with high probability, all the nodes adopt this maximum within the remaining COV rounds. Since no resets could be initialized after round i + COV (w.h.p.), this maximum must be between 2COV − (j − i) and 3COV − (j − i).

Corollary 4.2. A distributed reset can be performed in a complete or bounded degree graph in a self-stabilizing fashion with high probability within O(log n) rounds while using O(n log n) messages.

Our reset procedure works in any graph when substituting the appropriate COV value. For example, in bounded degree networks COV = O(D + log n), where D is the graph diameter [25].

4.1 The unison problem

Since the reset protocol presented in the previous chapter guarantees that, with high probability, the reset happens at the same round for all the nodes, it extends naturally to solve the unison problem. In a synchronous network
all the clocks advance at the same rate, and the reset protocol allows us to set all of them at the same time to some predefined value (e.g., 0), therefore all that is missing is a mechanism for detection of clock skew. It is not hard to see that for an efficient detection it is sufficient that every node checks its clock value against a random neighbor every round.

**Theorem 4.3.** The unison problem can be solved in a complete or bounded degree graph in a self-stabilizing fashion with high probability within \(O(COV + \log n)\) rounds while using \(O(n(COV + \log n))\) messages.

**Proof.** In order to solve the unison problem, the reset algorithm is used as follows. Every processor sends one message to a random neighbor every round. The message contains both its clock value and its reset value. A processor that receives a message with a clock value different from its own initializes a reset. When a processor resets, it sets its clock to a predefined value (e.g., 0).

When not all the clocks are synchronized, at least one node has a minority clock value. At least half of the other nodes will initialize a reset if they receive a message from this node, so the probability of reset in the case of the complete graph is at least 1/2. In the case of the bounded degree graph, this probability is at least 1/\(\Delta = O(1)\), because there must be two neighboring nodes whose clocks differ. Therefore, a reset will be initiated after \(O(\log n)\) rounds with high probability. When a reset is initialized, all the nodes are guaranteed to execute it simultaneously within \(O(COV)\) rounds with high probability, and set their clocks to a common value. Since every node sends one message each round, the message complexity of this procedure is \(O(n(COV + \log n))\) w.h.p..
Chapter 5

Spanning tree algorithm

5.1 Tree recoloring

We use the technique of repeatedly recoloring trees (see, e.g., [2, 4, 19, 21]) to allow the roots to detect a situation when there are several trees with the same treeid in the graph. Repeated tree recoloring works as follows: from time to time, the root of the tree chooses a new random color to use and sends it to its children. Propagation with feedback (see, e.g. [40]) is then used to make sure the whole tree is colored in this color: every node in the tree signals to its parent when its subtree has been completely recolored based on the signals from its direct children. When the root has received the signals from all its direct children, it can choose a new random color. Let CAP be the time required for the leader forest to capture all the nodes in the graph with high probability (as in Lemma 3.3). Recall that CAP is $O(\log n)$. Lemma 5.1 ensures that a new color is chosen at least once every $O(\log n)$ rounds:

Lemma 5.1. With high probability, at any time after the first $i \geq CAP$ request rounds, the height of every tree in the graph (max(treedepth)) is at most CAP.

Proof. Define the restricted leader forest $\mathcal{F}$ to be the subgraph of the leader forest that contains only the nodes with treedistance less than the last request round number. During the first request and confirmation rounds, those are the nodes that have the maximum treeid and are roots. During
the second request and confirmation rounds, those that have the maximum \texttt{treeid} and are either roots or their immediate children, etc.

We would like to prove that \( F \) captures every node in the graph within \( \text{CAP} \) rounds. In order to do so, consider the leader forest \( F' \) that would result from increasing the \texttt{treeid} of all the original leader forest roots by 1. Clearly, Lemma 3.3 applies to \( F' \).

\( F \) and \( F' \) contain the same nodes during the first round. A node in \( F' \) is always able to capture any node not in \( F' \). Additionally, a node is in \( F' \) after round pair \( i \) if and only if it either was in \( F' \) before, or received a request message from one of the nodes in \( F' \) during that round pair. Observe that the same holds for \( F \). We can therefore create a coupling between the random choices of the nodes in \( F \) and \( F' \), so that the same selections are made with the same probabilities and the same results. Therefore, during all the rounds, the distribution of \(|F|\) is the same as the distribution of \(|F'|\), and Lemma 3.3 applies to \( F \) as well.

We conclude that after \( \text{CAP} \) request rounds, all the nodes in the graph are in \( F \) with high probability. After \( \text{CAP} \) request rounds, the height of any tree in \( F \) is at most \( \text{CAP} \) by the definition of \( F \), and after a node is captured by one of the leader trees, its height can only decrease.

Because of Lemma 5.1 we only allow the \texttt{treedistance} of any node to be less than or equal to \( \text{CAP} \) (the value we take from Lemmas 2.6 and 2.8). This way we avoid very deep trees set up by the adversary. Any node that discovers that its \texttt{treedistance} is greater than this value resets it to 0 and becomes a root. Since in the proof neither \( F \) nor \( F' \) had any nodes with \texttt{treedistance} > \( \text{CAP} \), this modification does not affect the ability of the leader forest to capture every node in the graph within \( \text{CAP} \) rounds w.h.p..

The next corollary follows from this result and the fact that the time required to recolor a tree is at most twice its depth:

**Corollary 5.2.** Completely recoloring any tree in the leader forest using propagation with feedback requires at most \( O(\log n) \) rounds.

### 5.2 Competitor detection and synchronization

Let \( \text{CAP} = O(\log n) \) be the time required for a single root with the highest \texttt{treeid} to capture all the nodes, as in Lemma 3.3. After \( \text{CAP} \) rounds
have passed, with high probability there are multiple roots with the highest \texttt{treeid} if and only if there are multiple trees. In order to detect the presence of multiple trees, we use tree recoloring combined with a round counter. In our algorithm, only the roots perform the detection based on the request messages they receive. Whenever a root receives a request message from a node colored differently than its last two colors, that originating node must belong to a different tree. If a root receives such a message after enough rounds have passed, it starts a global reset procedure, which we specified in Chapter 4.

The purpose of the reset is for all the nodes to choose new random \texttt{treeids} simultaneously. We use the ID selection procedure from [3] to ensure that after every such reset, an unique leader exists with high probability regardless of $n$:

\textbf{Lemma 5.3} (Lemma 6 from [3]). \emph{If $n$ nodes run the procedure Choose(treeid), there is a unique node with the largest treeid, with probability at least $1 - \epsilon$, where $\epsilon = 1/r$.}

For our purpose, we take $r = n^\alpha$. The procedure consists of drawing two random integers as the \texttt{treeid}: the primary key is drawn from a Geometric distribution with $p = 1/2$, and the secondary key is uniformly selected from $[1, 36r \log(4r)]$. The largest primary key is less than $(\alpha + 1) \log n$ with high probability, however the secondary key may be as large as $36n^\alpha \log(4n)$, and therefore the size of a node \texttt{treeid} is $O(\log n)$ bits.

\textbf{Lemma 5.4}. \emph{If there are $R > 1$ trees in the leader forest, then the probability that no root receives a request message from a competing tree over $\text{CAP} + \log n$ request rounds is less than $2n^{-1}$.}

\textit{Proof.} By Lemma 3.3, the probability that the the leader forest has not absorbed all the nodes after $\text{CAP}$ request rounds is less than $n^{-1}$.

If all the trees belong to the leader forest, as long as no detection occurs, \textit{R} cannot change. At least $n(d - 1) + R$ request messages are sent every request round (recall the proof of Lemma 3.3), and each one of them arrives at one of the roots of a different tree with a probability $(R - 1)/n$. The probability that no such event occurs during the remaining $\log n$ request
rounds is:

\[
\left(1 - \frac{R-1}{n}\right)^{(n(d-1)+R)\log n} < e^{-(d-1)(R-1)\log n} \leq n^{-(d-1)(R-1)} \leq n^{-1}
\]  

(5.1)

The probability that any of those two unfavorable events occur is therefore at most \(2n^{-1}\).

Let the competition detection mechanism use \(\Omega(n)\) colors. The probability that a message arriving at a root from a different tree has the roots current or previous color is \(O(1/n)\). Divide the execution into epochs of \(\text{CAP} + \log n = O(\log n)\) request rounds. The following corollary follows from Corollary 5.2 combined with Lemma 5.4.

**Corollary 5.5.** If there is more than one root in the leader forest, and the detection mechanism is using \(\Omega(n)\) colors, then a reset is triggered within \(O(\log n)\) request rounds with high probability.

The detection of the lack of synchronization between the nodes’ request rounds is even easier. It is enough that a node receives a request message after its confirmation round or vice versa. Alternatively, we could reduce the synchronization to unison and use Theorem 4.3.

**Lemma 5.6.** If not all the nodes are synchronized, this is detected by some node w.h.p. within \(O(1)\) rounds.

**Proof.** Let there be \(cn\) nodes that consider the current round to be a request round. Assume that \(c \leq 1/2\), otherwise wait for the next round.

The \(cn\) nodes send at least \(cn(d-1)\) messages to random nodes during their request round. For each of these messages, the probability of triggering detection is \(1 - c\). Therefore, the probability that none of the \(cn(d-1)\) messages trigger detection is \(e^{cn(d-1)}\). By taking the derivative in \(c\), there is one minimum at \(c = 1/e\). Since \(c \in [1/n, 1/2]\), \(e^{cn(d-1)} \leq \max\left(n^{-(d-1)}, 2^{-\frac{d}{2}(d-1)}\right) = O(n^{-1})\) After \(\alpha\) rounds it will be \(O(n^{-\alpha})\).

### 5.3 Algorithm description and analysis

The per-node pseudocode for the final spanning tree algorithm appears on Figures 4–7. The local state variable \textbf{roundcounter} counts the number of rounds since the beginning of the algorithm or since the last reset.
**Figure 4** Final spanning tree algorithm, main loop

1: loop
2: \( R \leftarrow \) received request messages
3: \( C \leftarrow \) received confirmation messages
4: \( T \leftarrow \) received reset messages
5: run reset handler procedure (Figure 5)
6: if request then
7: \( \text{if } (R \neq \emptyset) \land (\text{reset} = 0) \text{ then } // \text{ not synchronized} \)
8: \( \text{reset} \leftarrow 2COV \)
9: end if
10: run request procedure (Figure 6)
11: else
12: \( \text{if } (C \neq \emptyset) \land (\text{reset} = 0) \text{ then } // \text{ not synchronized} \)
13: \( \text{reset} \leftarrow 2COV \)
14: end if
15: run confirmation procedure (Figure 7)
16: end if
17: request \( \leftarrow \neg \text{request} \)
18: end loop

**Figure 5** Final spanning tree algorithm, reset handler procedure

1: if \( T \neq \emptyset \) then
2: \( \text{reset} \leftarrow \max(\text{reset, max}(T)) \)
3: end if
4: if \( \text{reset} > 0 \) then
5: \( \text{reset} \leftarrow \min(\text{reset, } 2COV) - 1 \)
6: if \( \text{reset} = 0 \) then
7: \( \text{R, C} \leftarrow \emptyset \)
8: \( \text{roundcounter} \leftarrow 0 \)
9: \( \text{treeid} \leftarrow \langle \min(\text{Geo}(\frac{1}{2}), (\alpha + 2) \log n), 1 + [U(36on^\alpha \log(4n))] \rangle \)
10: request \( \leftarrow \text{true} \)
11: else
12: send \( \text{reset} \) to a random neighbor
13: end if
14: end if
Figure 6 Final spanning tree algorithm, request procedure

1: \( \text{roundcounter} \leftarrow \min(\text{roundcounter} + 1, \text{CAP}) \)
2: \( \text{coloringdone} \leftarrow \exists \ m \in C : ((m.\text{color} \neq \text{color}) \lor (m.\text{coloringdone} = \text{false})) \)
3: \( \text{children} \leftarrow \{\text{confirm.source} : \text{confirm} \in C\} \)
4: \( \text{while} |\text{children}| < d \text{ do} \)
5: \( \text{children} \leftarrow \text{children} \cup \text{random neighbor} \)
6: \( \text{end while} \)
7: send request \( \langle \text{treeid, treedepth, color} \rangle \) to the first \( d \) children

Figure 7 Final spanning tree algorithm, confirmation procedure

1: \( \text{candidates} \leftarrow \{r \in R : (r > \langle \text{treeid, treedepth} \rangle) \land (r.\text{treedepth} < \text{CAP})\} \)
2: \( \text{if} |\text{candidates}| > 0 \text{ then} \)
3: \( \text{parent} \leftarrow \max(\text{candidates}) \)
4: \( \text{treeid} \leftarrow \text{parent.treeid} \)
5: \( \text{treedepth} \leftarrow \text{parent.treedepth} + 1 \)
6: \( \text{if} \ \text{color} \neq \text{parent.color} \text{ then} \)
7: \( \text{color} \leftarrow \text{parent.color} \)
8: \( \text{coloringdone} \leftarrow \text{false} \)
9: \( \text{end if} \)
10: send confirmation \( \langle \text{color, coloringdone} \rangle \) to \text{parent.source} \)
11: \( \text{else} // \text{we are orphaned, become a root} \)
12: \( \text{treedepth} \leftarrow 0 \)
13: \( \text{if} (\text{roundcounter} \geq \text{CAP}) \land (\exists r \in R : (r.\text{color} \notin \{\text{color, oldcolor}\})) \land (\text{reset} = 0) \text{ then} \)
14: \( \text{reset} \leftarrow 2\text{COV} \)
15: \( \text{end if} \)
16: \( \text{if} \ \text{coloringdone} \text{ then} \)
17: \( \text{oldcolor} \leftarrow \text{color} \)
18: \( \text{color} \leftarrow \text{random color} \)
19: \( \text{end if} \)
20: \( \text{end if} \)
Note that in order to achieve bounded memory/message size, we must limit the values of the drawn treeids. This also makes our algorithms more practical to implement. The probability that any node draws a value larger than \((\alpha + 2) \log n\) for its primary treeid key in \(O(\log n)\) attempts (resets) is just \(o(n^{-\alpha})\). Therefore, with high probability, the execution of the algorithm with truncated entries is identical to its execution if the treeids weren’t limited. When the execution is different, it leads to \(tO(\log n)\) rounds increase in time complexity with a probability of \(tn^{-\alpha-1}\), therefore the expectation of this increase is \(O(1)\).

All the resets directly introduced by an adversary must cease within \(O(\log n)\) rounds. It does not matter that some nodes may reset earlier than others, or that the reset breaks some of the invariants we relied on earlier. After the first reset initialized by the algorithm (if such is needed), all the nodes are synchronized. After every such reset, with high probability there exists only one leader tree (Lemma 5.3). Therefore, there are only \(O(1)\) resets with high probability. By Theorem 4.1, every reset takes at most \(O(\log n)\) rounds. The time between resets is also \(O(\log n)\) because of Corollary 5.5 and Lemma 5.6. After the last reset, there is only one node with the highest treeid, so by Lemma 3.2, a spanning tree is constructed within \(O(\log n)\) rounds.

Every round, at most nd messages are sent. The size of all the variables is \(O(\log n)\) bits. As a result, we have the following theorem:

**Theorem 5.7.** On average and with high probability the spanning tree algorithm converges within \(O(\log n)\) rounds and uses \(O(n \log n)\) messages. The algorithm uses \(O(\log n)\) memory bits at every node, and this is also the size of its messages.

---

1The reader may note that whenever reset is greater than 0, it could make sense for the node not to run any procedures other than the reset handler. One may argue that since the node knows that it will reset in a finite number of steps, maintaining its state until then is meaningless. We choose to continue running the full algorithm for two reasons. First, the algorithm should try and provide “best effort” service as much as this is possible. Second, this optimization would introduce non-uniformity in the network load, which we are trying to avoid as was explained in Chapter 1.
Chapter 6

Conclusion and directions for further work

An interesting question is whether there is a gap in the communication efficiency between randomized protocols and deterministic ones. [1] showed a lower bound of $\Omega(n \log n)$ messages for any deterministic (even non self-stabilizing) leader election. However, in their model not all the processors may be initially active.

Our reset protocol works in any graph. It is possible that our other algorithms can be configured or improved to work in a more general topology as well (expander graphs look particularly promising). Some of this work seems extensible to asynchronous networks, or at least asynchronous bounded networks [12].

The analysis presented here can probably be extended to include crash failures. Crashed nodes do not respond to messages and are not required to be a part of the spanning tree. Recall that the rate of convergence of our algorithms depends on the rate of successful selections, i.e., random choices that lead to capture of new nodes or dissemination of information to them. It is easy to see that by crashing $\gamma(n)$ nodes, the adversary decreases the chance of success for every selection independently by a factor of $1 - \frac{\gamma(n)}{n}$. Therefore, in expectation, our algorithms should have to make $1 - \frac{\gamma(n)}{n}$ times as many selections to achieve. On the other hand, the number of successful selections the algorithm needs to make is also reduced by a factor of $1 - \frac{\gamma(n)}{n}$. We could therefore look at this problem as being equivalent to the original
with $n' = n - \gamma(n)$ and additional slowdown factor at most $1 - \frac{\gamma(n)}{n}$ in expectation.

Another direction for improvement is a tighter mathematical analysis. The reduction in Chapter 2 fails to capture the intuition that re-capturing an existing sub-tree of a new node should be easier than hitting new nodes randomly. After running some simulations (see Figure 8), it seems to have a real effect on the algorithm’s time complexity. Unfortunately, we were unable to find a method that would allow us to account for its effects in the theoretical analysis. We hope that a further investigation into this random process may lead to discovery of new methods of analysis for distributed randomized algorithms.

An interesting direction would also be to decrease the number of random bits required by the algorithm. For example, [17] shows that the results of [39] can be achieved with a reduced use of randomness.
Bibliography


In Chapter 3, the randomized process is determined by the fact that the algorithm successfully builds a tree. For each vertex, we consider only the randomized color.

In Chapter 4, the computation of the algorithms is presented. In unison communication, the algorithm also presents. With the algorithm, gossip communication is used. The gossip communication takes place in a number of rounds, each of which is a single vertex. After each round, the vertex sends a message to all its neighbors. The gossip communication is used in parallel with the randomized process. The gossip communication is used in a number of rounds, each of which is a single vertex. After each round, the vertex sends a message to all its neighbors. The gossip communication is used in a number of rounds, each of which is a single vertex. After each round, the vertex sends a message to all its neighbors.
כתכיזי מורגש

לפי קרי 20 שנה ב-2011 במעבדה מחשבים המזくださה הטובה ביותר במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנولوجי...

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב.

בחלק מה_pv ב-2011 במעבדה מחשבים המז предостו תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המחשב ע"ש שלמה. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המחשב. במקביל, ייעודו במעבדה מחשבים הוא שילוב תכניות מחקריות ופיתוח בטכנולוגיה במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشاهد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشدد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشدد, אוניברסיטת תל אביב, וב tho בתוכנית מחקר במדעי המشدد, אוניברסитет...
המחק נועה נעשתה הברגהית פרופسور ש כי מומכיקו חותשים וניהול בפקולחה למידע המחשב.

הראות, ברטון להורדה למוטשחתיה והביר לע הניהובה והמתכונה של הפקולחה סיפק לי במידח השכינה. אני מודע למלאתה שלל פרופסור ש כי קוקית מוצגạ לע הניהובה בסיסיתاعدת מתכונה והפקולחה הגדישה אפלו.

אכשף המינים ולא לח קולות המחקה היה לא היה יתאמה של הניהובה. כי אני מודע למיתר עליים ממחזרי בין גבורי על לשונת רכוב על שמחת מעניינים, במחדה בלשיבו המוקדמים של המחקה.

למודי בוכננים סיפק לי הניהובה תלולך לא מעורר אין我又 ידוע ובצורה מעין כנה מוהם. אני מודע למאמריהם תנית עטיה וסוכנותיبخ בבית ההכרתי בת החום, וגס לכבודו יספ שספוף

ש loginUser הנית לאפאזוות לממדות לממדות טמוון מייממים מעניינים.

אני מודע את המחקה הזה ל交流合作 שלי, מחא פוליקהו.

אני מודע לתקני הקוקית הלאומית למידע על התכונה הכיספת המיורבבה ביהותולמות.
iership

לשם مليי חلة עם הדרישה של הקבוצה המרכז
מניסים למטעים במדעי המחשב

דmitriy יונקו

רומש לסטט הטכניון - מרכז טכניות לישראל
טבת תש"ע' א חיפה דצמבר 2010
ייצוב עצמי עם תכשורת ייעלת

דמיטרי זינקוק