Towards a Theory of Peer-to-Peer Computability

Joachim Giesen Roger Wattenhofer Aaron Zollinger
{giesen,wattenhofer,zollinger}@inf.ethz.ch
Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland

Abstract

A peer-to-peer system is a distributed system with no physical or logical central authority. We give a formal model of a peer-to-peer system where agents communicate through *read-modify-write* registers that can be accessed by exactly two agents. For this model, we study so-called ordering decision tasks for wait-free agents. We show how agents can determine their position in a total linearizable order in the peer-to-peer model. We also show that electing a leader among the agents and finding a predecessor agent in the total ordering cannot be implemented without a central authority. Our peer-to-peer model is related to other models of distributed computing, specifically to concurrent objects in asynchronous shared memory and to switching (counting) networks.

1 Introduction

After a first wave of successful Internet applications, such as electronic mail or the World Wide Web, we are currently witnessing a second wave of distributed applications, which are often dubbed "peer-to-peer." Peer-to-peer computing is the sharing of computer resources and services by direct exchange between client systems. These resources and services include the exchange of information (Napster, Freenet, etc.), processing cycles (distributed.net, SETI@home, etc.), or disk storage for files (OceanStore, Farsite, etc.). It is anticipated that future middleware, groupware, or service platforms might also be built in a peer-to-peer manner.

Current algorithmic peer-to-peer computing research focuses on content location service. Data structures that allow fault-tolerant non-centralized distributed dictionary services are studied [KLL+97, PRR97, KBE+00, DR01, RFH+01, RRVV01, SMK+01]. These services are at the heart of any file sharing application such as Napster. However, file sharing is a simple form of a peer-to-peer application. Major industry leaders are currently promoting their "platform for services; the next generation of software that connects our world of information, devices and people." These Internet based distributed services will need organization and coordination structures beyond content location and distributed dictionaries. In this paper we therefore study fundamental coordination primitives that are all-time favorites in distributed computing.

There is no clear consensus on what peer-to-peer is[1]. In this paper we think of a peer-to-peer system as a distributed system with no (physical or logical) central authority. In the next section we will present a formal model of computation, intuitively, however, our peer-to-peer system is as follows. In the peer-to-peer system there are agents (processes, nodes) and registers (communication channels, memory cells). Agents can only communicate through registers. Each pair of agents has a private register that cannot be accessed by any other agent. There is no register that is shared by more than two agents. When visiting a register, an agent reads the content in the register and writes new content into the register in one atomic step. The write depends on the read; it is allowed that an agent stores its whole state (after the read) in the register.

Agents want to solve a decision task, for example electing a leader among them. We are interested in "wait-free" implementations [Her91], that is, an agent wants to decide definitively after a finite number of steps (visits of registers) without waiting for other agents. Agents are asynchronous: The time be-

[1] Some argue that peer-to-peer computing is a post-client/server paradigm where every client is also a server and vice versa (Gnutella names a hybrid client/server node a "servent"), sometimes it is stressed that all those nodes need equivalent computational power. Often it is argued that peer-to-peer computing takes advantage of existing desktop computing power and networking connectivity, allowing off-the-shelf clients to leverage their collective power beyond the sum of their parts. Others even believe that the most important aspect of peer-to-peer computing is a post-IP and post-DNA addressing system (Clay Shirky, What Is P2P... And What Isn't?, *www.openp2p.com*). Our model captures some of these aspects, but they remain in the background.

tween two register visits is unbounded [AW98]. Also, there is no global starting time for agents; it is possible that an agent completes all its steps and therefore chooses its value of the decision task before another agent even "wakes up".
In this paper we focus on ordering decision tasks. The agents try to totally order themselves; however, no agent needs to know the whole order. Instead, each agent only needs to have a partial view of the total order. Examples for ordering decision tasks are:

- *Leader Election*: each agent needs to know the first agent in the total order
- *Position*: each agent needs to know its position in the total order
- *Predecessor*: each agent needs to know its predecessor agent in the total order

To rule out trivial solutions, we want the total order to be "linearizable" [HW90]: If in a particular execution an agent A has never directly or indirectly heard of agent B, agent A needs to have a lower position than agent B in the total order.
For the reader familiar with the rich and well-studied asynchronous shared memory model[2] our peer-to-peer model can be expressed as one of its specializations. Agents are called processes. We allow registers to be of the strongest *read-modify-write* type: A process can read, compute a function of the read value, and write the value of the function to the register, all in one atomic step. However, each "peer-to-peer" register is only accessible by exactly two processes.
A peer-to-peer implementation for only 2 agents is trivial because both agents can access the same register. The model becomes interesting with 3 agents, so we will first show all results for the simpler case of 3 agents and then apply them to larger systems.
This paper is organized as follows. Section 2 defines a formal model of peer-to-peer computation. In Section 3 we define the structural basis on which we prove in Section 4 that *Leader Election* and *Predecessor* cannot be achieved in the peer-to-peer model, whereas *Position* can. We will prove these results with a variety of different techniques: topological/structural (similar to [HS99]), by construction, and by reduction. In Section 5 we describe the relationship between our peer-to-peer model and switching networks. We conclude the paper in Section 6 with a summary, a discussion on the relation of our peer-to-peer model to other computational models, and open problems.

[2]The other major model for distributed computing—message passing—will be discussed in Section 6.

2 Formal Peer-to-Peer Framework

The purpose of this section is to introduce the notation we are going to use. The main entities we consider are agents and registers. Agents can write into or read from registers to which they have access.

Agent. The peer-to-peer framework contains a finite number of agents. Each agent is assigned a unique identity from the set $\mathcal{A} = \{1, \ldots, n\}$.

Register. Registers are of *read-modify-write* type. That is, a read and a write operation can be combined into one atomic operation where the write can depend on the read operation.

Each pair of agents shares a *read-modify-write* register. Thus we have $\binom{n}{2}$ registers in total. We denote the register shared by agents i and j, $i < j$, as r_{ij}.

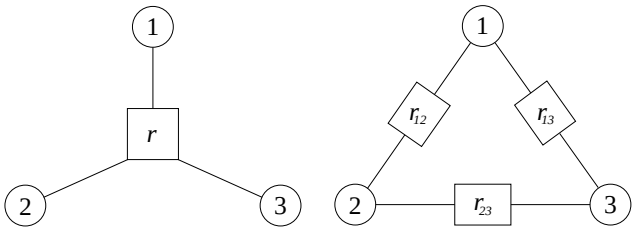


Figure 1: In the classical model of computation, depicted on the left, all agents can access all registers. The right figure represents our peer-to-peer model, where each register can only be accessed by two agents.

Register access. We denote an access of agent i to the register shared with agent j by the pair (i, j).

Agents access the registers within their reach in a certain order. We call this order a strategy.

Strategy. The strategy S_i of agent i forms a tree. The nodes of the tree are accesses of the form $(i, \cdot)$. The children $(i, j_1), \ldots, (i, j_k)$ of a node (i, j) denote the possible accesses performed after accessing register r_{ij}. Which of the registers $r_{ij_1}, \ldots, r_{ij_k}$ is accessed

next depends on the access (i, j). If the tree degenerates to a sequence, the strategy is called oblivious. Otherwise it is called non-oblivious.

Lemma 1 *A non-oblivious strategy can always be simulated by an oblivious one.*

PROOF. Consider the strategy of agent i. All nodes in the strategy tree are of the form $(i, \cdot)$. The depth of a node in the tree is the number of edges in the path that connects the node with the root of the tree. Assume that the maximum depth of a node in the tree is k. We construct a sequence from the strategy tree as a concatenation of rounds $R_0, \ldots, R_k$ as follows: R_j is a set containing all different accesses in nodes of depth j. The accesses in round R_j can be ordered arbitrarily. R_0 is for instance the round of size one that contains only the access in the root of the strategy tree. The concatenated rounds define an oblivious strategy that is sufficient to simulate the original strategy of agent i. In every round, i only considers the information gathered in one access; the choice of this access depends on the information seen in the previous rounds. □

In the following we assume that all protocols are oblivious.

EXAMPLE 1: OBLIVIOUS STRATEGIES FOR THREE AGENTS.

Agent 1: $(1,2), (1,3), (1,2)$
Agent 2: $(2,1), (2,3), (2,1)$
Agent 3: $(3,2), (3,1), (3,2)$

Protocol and execution. Every agent follows its own strategy. All these individual strategies together define an access protocol $\mathcal{P}$ of the agents. The protocol is a class of access sequences. Every such sequence contains all accesses of every agent and the accesses of the agents are ordered as prescribed by the agent protocols. We call an access sequence $A \in \mathcal{P}$ an execution.

EXAMPLE 2: THREE EXECUTIONS OF THE PROTOCOL DEFINED BY THE STRATEGIES OF EXAMPLE 1.

(1,2),(1,3),(1,2),(2,1),(2,3),(2,1),(3,2),(3,1),(3,2)
(1,2),(1,3),(1,2),(2,1),(2,3),(3,2),(2,1),(3,1),(3,2)
(1,2),(1,3),(1,2),(2,1),(3,2),(2,3),(2,1),(3,1),(3,2)

We associate with every execution $A \in \mathcal{P}$ a directed acyclic graph (DAG). The DAG captures all the information of A that we need later on. Essentially, the execution DAG captures how information flows between the agents.

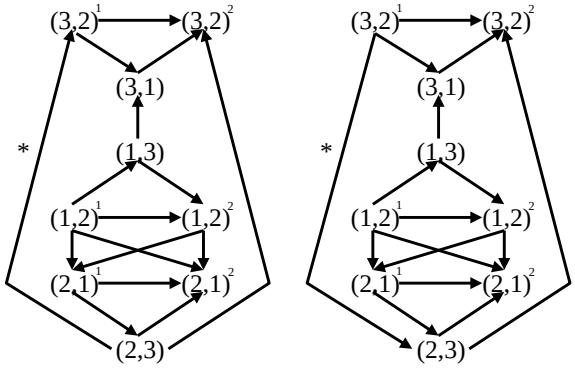
Execution DAG. Given an execution $A \in \mathcal{P}$. Let $A = a_1, \ldots, a_m$. The vertices of the execution DAG are the accesses in A. A pair $(a_l, a_{l'})$ defines an edge of the DAG if and only if $l < l'$ and one of the two conditions holds

(1) a_l and $a_{l'}$ are of the form $(i, \cdot)$.

(2) a_l is of the form (i, j) and $a_{l'}$ is of the form (j, i).

The edges induced by the first condition reflect the ordering in the strategies of the agents. The edges induced by the second condition reflect the ordering of the accesses of two agents to their common register in the execution A.

EXAMPLE 3: DAGS THAT CORRESPOND TO THE EXECUTIONS IN EXAMPLE 2. REPEATED ACCESSES TO ONE REGISTER IN THE SAME STRATEGY ARE IDENTIFIED BY SUPERSCRIPT NUMBERS. THE LEFT DAG CORRESPONDS TO THE FIRST TWO EXECUTIONS, THE RIGHT ONE TO THE THIRD EXECUTION. NOTE THAT THE ONLY DIFFERENCE BETWEEN THE TWO DAGS CONSISTS IN REVERSAL OF THE MARKED (*) EDGE.



Task. A task function

$$t : \{1, \ldots, n\} \to \{1, \ldots, n\}$$

assigns to every agent the identity of another agent. A task is a class of task functions. The task can be defined by constraints which itself can depend on the execution of a protocol. The agents perform a task

if they compute a task function in the prescribed class of task functions.

3 Protocol Graph

The execution DAG carries all the information we need about an execution. In this section we use the execution DAG to define a protocol graph which we will use to prove results in our peer-to-peer model of computation. To define the protocol graph we use an indistinguishability relation on $\mathcal{P}$, the set of all executions of a protocol. This relation describes whether two executions can be distinguished by the agents. We call two executions *indistinguishable for agent i* if i cannot distinguish which one of the two executions is actually performed. We call two executions simply *indistinguishable* if none of the agents can distinguish which execution is performed.

Indistinguishability relation. Two protocol executions $A, B \in \mathcal{P}$ are indistinguishable if and only if they have the same associated execution DAG. If A and B are indistinguishable we write $A \sim B$. Indistinguishability is an equivalence relation on $\mathcal{P}$, that is, the following holds:

(1) $A \sim A$ for all $A \in \mathcal{P}$ (reflexivity)

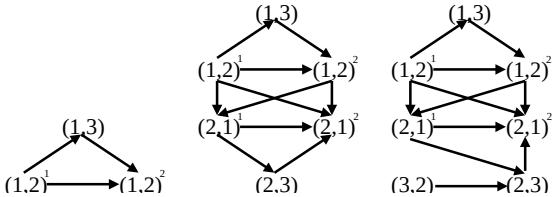
(2) $A \sim B \Rightarrow B \sim A$ (symmetry)

(3) $A \sim B \wedge B \sim C \Rightarrow A \sim C$ (transitivity)

For every agent we prune the DAG and use the pruned DAG to define an equivalence relation on $\mathcal{P}$ for every agent.
A vertex a_l of the DAG is reachable from vertex $a_{l'}$ if there is an oriented path in the DAG from $a_{l'}$ to a_l. For agent i the pruned DAG retains all vertices a such that there is a vertex of the form $(i, \cdot)$ reachable from a.
Informally speaking, the pruned DAG describes the information flow relevant to agent i. An information flow path is removed if i does not learn about it.
We call two executions $A, B \in \mathcal{P}$ indistinguishable for agent i if the pruned execution DAGs associated with A and B are identical. If A and B are indistinguishable for agent i, we write $A \sim_i B$. Indistinguishability for agent i is also an equivalence relation on $\mathcal{P}$.

Now we are prepared to define the protocol graph for some protocol $\mathcal{P}$.

Example 4: Pruned DAGs from Example 3. Pruning both DAGs from Example 3 with respect to Agent 1 results in the same DAG with three vertices (on the left). Pruning with respect to Agent 2 results in two different DAGs (in the middle and on the right).



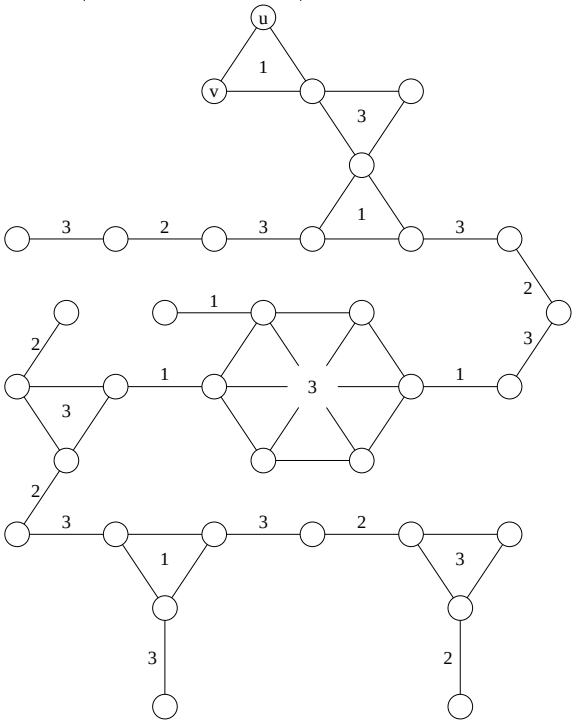
Protocol graph. The protocol graph $G = (V, E)$ is a labeled multigraph. It consists of a vertex set V and a set of labeled edges E defined as follows:

Vertices V:	equivalence classes of the set $\mathcal{P}$ with respect to the indistinguishability relation $\sim$
Edges E_i:	two vertices u and v define an edge $\{u, v\}$ with label i if there exists $A \in u$ and $B \in v$ such that $A \sim_i B$

Lemma 2 *The protocol graph G for $n > 2$ is connected.*

Proof. Let $A \in \mathcal{P}$ be an arbitrary execution consisting of m accesses $a_1, a_2, \ldots, a_m$ and G be the protocol graph of $\mathcal{P}$. We prove this lemma by induction over the last $p < m$ accesses of A. Assume the subgraph of the total protocol graph formed by all legal permutations of $a_{m-p+1}, \ldots, a_m$ is connected. (A legal permutation is compliant with the strategies of all agents, that is, accesses by the same agent may not be permuted.) We can show that, given the above assumption, the subgraph formed by the legal permutations of the last $p+1$ accesses is connected. Together with the induction base at the end of the proof, it follows that the total protocol graph is also connected.
Consider the access a_{m-p} performed just before the p last accesses. We assume a_{m-p} and a_{m-p+1} are not executed by the same agent: Transposing a_{m-p} and a_{m-p+1} will result in a legal execution. (If both of those accesses *are* performed by the same agent, another access can be moved to the front by making use of the induction assumption. If the last p accesses of A exclusively contain accesses by the same agent, the

Example 5: Protocol graph of the protocol defined by the strategies of Example 1. A total of 1680 possible executions are partitioned into 34 vertices. The first two executions of Example 2 are elements of vertex u, whereas the third execution is contained in vertex v. The two vertices are linked with an edge labeled 1, since agent 1 cannot distinguish the contained executions (see Example 4).



considered protocol subgraph consists of one vertex containing exactly one execution, in which case the induction step works trivially.) We call A the execution before switching a_{m-p} and a_{m-p+1}, and B the execution after this transposition. When switching the places of a_{m-p} and a_{m-p+1}, we have to distinguish three cases:

- a_{m-p} and a_{m-p+1} are of the form $(i_1, j_1), (i_2, j_2)$ and $i_1 \neq j_2$ or $i_2 \neq j_1$, that is, i and j access two different registers. No agent can distinguish A and B, which are therefore elements of the same protocol graph vertex.

- a_{m-p} and a_{m-p+1} are of the form (i, j) and (j, i): Agents i and j access their common register. In addition, there is at least one agent $k \neq i, j$ which cannot distinguish the two executions; that is, there is no path in any of the DAGs of A and B from one of the accesses (i, j) or (j, i) to an access of the form $(k, \cdot)$. Since i and j can distinguish the executions due to their altered access order to the same register, A and B are elements of two different protocol graph vertices u and v. However, u and v are connected in G by an edge labeled k.

- a_{m-p} and a_{m-p+1} access the same register and all agents can distinguish the two executions. That is, a_{m-p}, a_{m-p+1} are of the form $(i, j), (j, i)$ and for all $k \neq i, j$ there is a path from one of the accesses $(i, j), (j, i)$ to an access $(k, \cdot)$ in the DAGs of A and B. All n agents can distinguish A and B, which lie in two vertices u and v. The protocol graph G therefore contains no edge between u and v. However, we can construct an execution A' as follows: We permute the last $p-1$ accesses in such a way that all remaining accesses performed by one "third party" agent $k \neq i, j$ occur directly after a_{m-p+1}. (Note that there exist such accesses $(k, \cdot)$ among the last $p-1$ accesses of A; otherwise, k would not learn about the transposition of a_{m-p} and a_{m-p+1}.) The execution B' now results from a transposition of accesses a_{m-p} and a_{m-p+1} in A'. k cannot distinguish A' and B', since neither i nor j will have an opportunity to inform k before its termination. According to the induction assumption, it is consequently possible to find a path A - A' - B' - B in G, that is, u and v are connected.

The analysis of the above cases shows that if the protocol subgraph formed by the legal permutations of the last p accesses is connected, it is possible to move a_{m-p} among the last p accesses without breaking connectivity of G. Since the accesses of the execution are chosen arbitrarily, it follows that the subgraph formed by the legal permutations of the last $p+1$ accesses of *any* execution is connected.

As to the induction base, transposing the last two accesses of any execution is perceived solely by the two involved agents, if at all. Since the total number of agents is greater than two, at least one agent cannot distinguish the two executions; this protocol subgraph is connected. □

4 Peer-to-Peer Computability

In this section we study three tasks: *Leader Election, Position,* and *Predecessor*. The functions in these tasks are constrained by a relation on $\mathcal{A}$ that we again derive from the execution DAG. That is, these tasks depend on the execution of a protocol. We will show that *Position* can be solved by the agents in contrast to *Leader Election* and *Predecessor*.
As the indistinguishability relation on $\mathcal{P}$, the invisibility relation on $\mathcal{A}$ is defined via the execution DAG. The invisibility relation indicates whether one agent is invisible for another agent in the execution A of protocol $\mathcal{P}$, i.e. it indicates if one agent can decide if another agent is alive or not.

Invisibility relation. Given $A \in \mathcal{P}$. Agent j is invisible for agent i if there is no oriented path in the execution DAG of A from an access of the form (j, i) to an access of the form (i, j). Invisibility is a relation on $\mathcal{A}$ that we denote by $\nleftarrow_A$; we write $i \nleftarrow_A j$ if agent j is invisible for agent i.

We use the invisibility relation to define three tasks the agents have to perform. All three tasks are ordering tasks, that is, the agents compute a total order over themselves according to the execution. This order forms the basis for the actual task computation. In the *Leader Election* task, all agents have to agree on the first agent's identifier. In the *Position* task, every agent finds its own position within the total agent order. In the *Predecessor* task, the first agent chooses its own identifier, whereas every other agent computes the identifier of its direct predecessor in the agent order.

Leader Election: (1) For all $i, j \in \mathcal{A}$: $lead(i) = lead(j)$.
(2) If $i \nleftarrow_A j$ then $lead(i) \neq j$.

Position: (1) If $i \neq j$ then $pos(i) \neq pos(j)$.
(2) If $i \nleftarrow_A j$ then $pos(i) < pos(j)$.

Predecessor: $pred(i) = \begin{cases} i & \text{if } pos(i) = 1 \\ j & \text{if } pos(j) = pos(i) - 1 \end{cases}$

In the following, we analyze the computability of these tasks in our peer-to-peer model using the structures defined so far.

Theorem 1 Leader Election *is impossible for* $n > 2$.

Proof. According to the definition of *Leader Election*, an execution beginning with all accesses of the form $(i, \cdot)$ before any other access results in $lead(j) = i$ for all agents j. Since the protocol $\mathcal{P}$ contains all executions, there is such an execution for all agents i. Translated to the protocol graph G of $\mathcal{P}$ this means, that G contains for each agent i at least one vertex whose executions result in $lead(j) = i$ for all j. Since a result is computed for every execution of $\mathcal{P}$, the vertices V of G are partitioned into n sets of vertices V_i, $1 \leq i \leq n$, where V_i denotes the set of all vertices containing executions which result in $lead(j) = i$ for all agents j. Since the protocol graph is connected, there exist edges connecting two such sets of vertices S_i and S_j, $i \neq j$. There is consequently at least one agent which cannot distinguish two executions $A \in S_i$ and $B \in S_j$ and will choose the same result in both executions. This contradicts the task definition, according to which in A all agents have to choose i and in B they have to choose j. □

Note that this result can also be found with an FLP-type proof [FLP85]. Such a proof would argue over a point in time, at which a decision as to the *Leader Election* result has to be taken. We think, however, that in this case, our proof based on complete executions yields deeper insight into the structure of the peer-to-peer model.

Theorem 2 Position *is possible for* $n = 3$.

Proof. For this theorem we give a constructive proof, defining a concrete protocol which solves the *Position* task for three agents. Consider the protocol with the agent strategies $S_1 = (1,3),(1,2),(1,3)$, $S_2 = (2,3),(2,1)$, and $S_3 = (3,2),(3,1)$. As shown in Figure 2, the protocol graph for this protocol allows assignment of a *Position* result order to each vertex of the protocol graph without conflicts over edges. Although, for each edge (u, v), the labeling agent, say i, cannot distinguish executions in u and v and therefore has to choose the same result for executions in both u and v, there is no conflict: In every such case, i assumes the same position in the result order of both u and v.
In other words, after executing the protocol each agent knows where the performed execution lies within the protocol graph, since, informally speaking, most executions are distinguishable. The few cases where single agents cannot distinguish executions from different vertices do not provoke any conflicts, as the respective agents can choose the same result for all indistinguishable vertices. In order to compute the result of the *Position* task

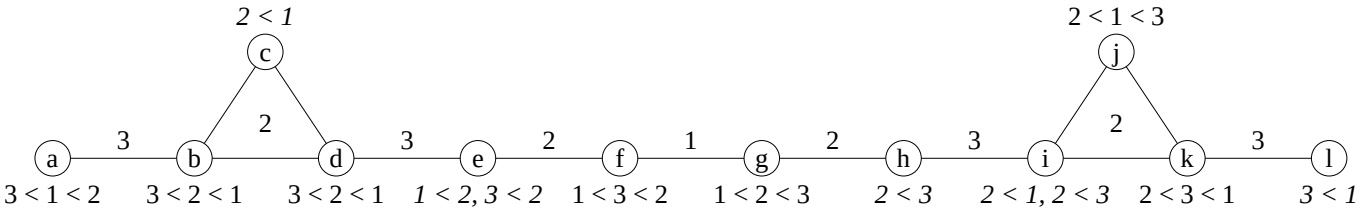


Figure 2: Protocol graph for 3 agents with strategies $S_1 = (1,3),(1,2),(1,3)$, $S_2 = (2,3),(2,1)$, and $S_3 = (3,2),(3,1)$. The vertices are labeled with the corresponding *Position* task results. $i < j$ is short for $pos(i) < pos(j)$. The conditions on the task result imposed by the invisibility relation define a total agent order for the executions in vertices a, b, d, f, g, j, and k. The invisibility relation only defines a partial agent order for vertices c, e, h, i, and l (*italics*). Note that for all of these vertices, a unique total agent order is inferred by the indistinguishability relation (edges). In particular, vertex c is attributed the same agent order as vertex b, h as g, i as j, and l as k, whereas vertex e assumes the order $3 < 1 < 2$.

function, each agent finds the vertex containing the current execution within the protocol graph and chooses its result according to the label of that vertex. This interpretation of the information gathered during the protocol execution can be considered a distributed algorithm. □

Theorem 3 Predecessor *is impossible for* $n = 3$.

Proof. We prove this theorem by reduction to the *Leader Election* task. Assume the three agents *can* solve the *Predecessor* task, that is, each agent knows which other agent is its predecessor in the agent order. Since the *Position* task is solvable for three agents, each agent can additionally—and in parallel to the computation of its predecessor—compute its own position within the order. Assume that the result of the execution is $pos(i) < pos(j) < pos(k)$. Agent i knows that it is the first agent. Agent j knows that it is in position 1 and that i is its predecessor and therefore the first agent. Agent k knows its position and its predecessor, and can infer from this information, that agent i must be the first agent, since $n = 3$. If each agent knows both its own position within the result order and its predecessor, it also knows which agent holds the first position: *Leader Election* is possible. As we have shown that *Leader Election* is impossible, our assumption on the *Predecessor* task must be wrong. □

Corollary 1 Predecessor *is impossible for* $n > 3$.

Proof. Assume there exists an $n > 3$ for which the solution of the *Predecessor* task is possible with a protocol $\mathcal{P}$ and some *Predecessor* computation algorithm α. Consider three agents i, j, and k and let $\mathcal{P}_{ijk}$ be the subset of all executions in which all accesses of the agents i, j, and k occur before any access by another agent. According to the order conditions imposed by the invisibility relation, all executions in $\mathcal{P}_{ijk}$ place the agents i, j, and k among the first three positions of the result order. Let i, j, and k taken together perform a total of m accesses. Note that any two executions $A, B \in \mathcal{P}_{ijk}$ with the same prefix of length m yield the same task result for agents i, j, and k, since the rest of the execution is immaterial to them.

We define an algorithm α' as follows: Use protocol $\mathcal{P}$ and algorithm α but let the three agents i, j, and k ignore all accesses performed by any agent other than i, j, and k. Using α', i, j, or k never see an access by any other agent $l \neq i, j, k$ and assume they terminate before l performs its first access. From the point of view of i, j, or k, any execution $A \in \mathcal{P}$ is indistinguishable from any execution $B \in \mathcal{P}_{ijk}$ whose prefix of length m corresponds to the relative order of the accesses by i, j, or k in A. i, j, and k compute the same result as with algorithm α for execution B, that is, they place themselves among the first three positions of the agent order and compute the correct *Predecessor* result for three agents.

Based on a protocol solving *Predecessor* for $n > 3$, we can solve the same task for three agents. Since, however, we have shown that *Predecessor* is impossible for three agents, there cannot exist any protocol solving this task for $n > 3$ agents. □

5 Switching Networks

In this section, we focus on the relationship between our peer-to-peer model and switching net-

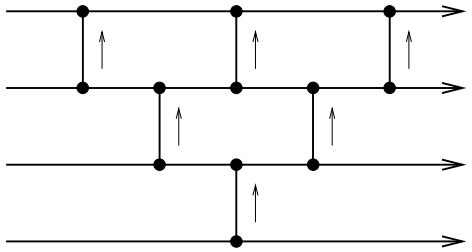


Figure 3: "Triangle" Counting Network

works. First, we describe switching networks in general. Then we show the correspondence between linearizable counting networks [HSW96] as a special type of switching networks and the solution of the *Position* task in our peer-to-peer model.

5.1 General Switching Networks

A switching network is a directed acyclic graph consisting of switches (vertices) and links (edges) connecting switches. Tokens are sent along the links from switch to switch. Each token enters the network at an input link—one out of a number of "open" edges without source switch—and leaves the network at an output link—output links being edges without destination switch. Switches—usually connecting several input links to several output links—decide according to a function where to route incoming tokens, that is, on which output link to forward a token received on an input link. This routing function can depend on a state internal to each switch. The depth of a switching network is the maximum number of switches a token has to traverse. For a detailed introduction on switching networks see for example [AHS94, BM96].

5.2 Counting Networks

Counting networks are a special class of switching networks. As shown in Figure 3, a counting network is usually drawn as a set of horizontal wires and vertical balancers, each one connecting two wires. Tokens enter the network on the left end of a wire and are routed to the right end of a wire, where they leave the network. Each balancer routes incoming tokens independently of the wire it arrives on; the first arriving token is forwarded to the balancer's upper output wire, whereas all subsequently arriving tokens are routed alternately to the lower and the upper output wire. Figure 3 shows a linearizable counting network as introduced in [HSW96], where balancers are positioned in a well-defined triangle-like structure. An arrow next to a balancer indicates its current routing direction; it is reversed after forwarding a token. Using the terminology of general switching networks, the balancers correspond to switches with two input and two output links and having a one-bit internal state deciding where to route incoming tokens. The wire segments between the single balancers correspond to the general switching network links.

5.3 Switching Networks and the Peer-To-Peer Model

The main idea of this section is to simulate switching networks with the peer-to-peer model. In particular, we study switching networks containing no balancer that routes more than two tokens. This implies that routing decisions by switches can be taken in a peer-to-peer way. All agents know the structure of the switching network; each token is represented by an agent. Particularly:

Theorem 4 *In the peer-to-peer model we can simulate any switching network with finite depth if no more than two tokens traverse any switch and if the switch function is computable in the peer-to-peer model.*

Proof. We will prove this theorem by showing how this simulation takes place in the peer-to-peer model. First, each token i entering the switching network is associated to an agent i. The execution of agent i will simulate how token i is routed through the network. Token i reaching a balancer b is simulated by agent i initiating a "sweep" $(i,1)\dots(i,i-1),(i,i+1)\dots(i,n)$ over all registers within its reach: In every register r_{ij}, agent i checks whether another agent j has already marked that its associated token has passed balancer b; if so, agent i concludes that token i is second, (and should be routed to the lower output wire of balancer b in case of a counting network). Otherwise, it writes information into register r_{ij}, signaling that balancer b has forwarded a token. If agent i completes this register "sweep" without finding information about any other token previously passing balancer b, it can conclude that token i is the first token arriving at balancer b (and will consequently continue on b's upper output wire in case of a counting network). Note that—except in special cases—agent i cannot predict which other token j is going to pass balancer b. □

It is proved in [HSW96, Theorem 4.10] that the REVERSED-SKEW is a linearizable counting network. For our model it is however sufficient to consider the simpler "triangle" counting network of Figure 3 with as many wires as there are agents in the peer-to-peer model. The "triangle" counting network is a linearizable counting network, that is, if a token exits on wire i, we can be sure that it has "seen" the tokens that exit on wires j with $j < i$. Both conditions required in Theorem 4 hold for the "triangle" counting network. We can therefore conclude directly:

Corollary 2 Position *is possible for* $n > 3$.

Note that in the peer-to-peer model not only counting networks but also more general switching networks can be simulated as long as the switch routing function is computable in the peer-to-peer model. If impossibility results can be proved for the peer-to-peer model, they can directly be transferred to these switching networks.

6 Conclusions

In this paper we have proposed a first theoretical model for peer-to-peer computability. We have shown that some ordering decision tasks, such as *Leader Election* or *Predecessor*, cannot be implemented in the peer-to-peer model, whereas others, such as *Position*, can. We have focused on ordering tasks, that is, tasks without inputs; it is clear, however, that also tasks with inputs can be studied. The *Consensus* decision task for example cannot be solved for the same reasons as *Leader Election*, and *Prefix Sum* (a. k. a. *fetch-and-add*) can be solved for the same reasons as *Position*.

Our peer-to-peer model of computation is related to other models in distributed computing. When we translate our model into asynchronous shared memory, our ordering decision tasks have the one-shot concurrent object equivalents *test-and-set*, *fetch-and-inc*, and *swap* [HW90]. In asynchronous shared memory one primary research issue is the assessment of the power of different register types. Herlihy has presented a hierarchy of more and less powerful types, on the basis of how many processes are required to reach consensus [Her91]. It is notable that all the concurrent objects we study in this paper have consensus number 2; however, a one-shot *fetch-and-inc* can be implemented whereas a one-shot *swap* cannot. The most studied asynchronous shared memory register type is the read/write register, where processes can either read or write a register atomically but not both. Israeli and Li [IL93] and Vitanyi and Awerbuch [VA86] have shown that a multi-writer multi-reader register can be simulated with single-writer single-reader registers. A single-writer single-reader register is clearly a peer-to-peer register, if however a "weak" one. In our work we show that the omnipotent—but peer-to-peer—*read-modify-write* register cannot simulate an omnipotent globally shared *read-modify-write* register (because we cannot solve *Leader Election* with peer-to-peer registers, but we can solve *Leader Election* with a globally shared register).

Clearly, our peer-to-peer model is a shared-memory variant. Instead of restricting the way to access a register we restrict who can access a register. Contention—one of the main issues in the context of shared-memory models [DHW97]—naturally becomes noncritical, since no more than two agents access one register.

Some of our results could also have been attained with FLP-type proofs [FLP85] by reasoning over points in time at which agents have to come to a decision. In contrast, our approach considers complete executions and is of topological/structural nature, similar to the theory introduced in [HR95, HS99].

In some sense, a message passing based model would better fit the peer-to-peer philosophy of having loosely coupled agents. However, message passing lacks a reliable and asymmetric communication channel. We could not find a message passing peer-to-peer model with such a rich structure as our peer-to-peer shared memory model. We feel that this is a challenging open research area, though.

In Section 5 we showed that our peer-to-peer model can simulate switching networks, first introduced to the distributed computing community in the form of counting networks [AHS94]. This means that impossibility results for the peer-to-peer model translate to switching networks. In the recent years, several papers with impossibility results (or lower bounds) for switching networks were presented [HSW96, FH01]. We feel that the peer-to-peer model gives simpler means to prove these negative results.

In this paper we concentrate on computability rather than on efficiency. How to compute *Position* or other decision tasks or to work with distributed objects efficiently (and without simulating a linear-sized network) is another interesting open research problem.

References

[AHS94] James Aspnes, Maurice Herlihy, and Nir Shavit. Counting networks. *Journal of the ACM*, 41(5):1020–1048, September 1994.

[AW98] Hagit Attiya and Jennifer Welch. *Distributed Computing: Fundamentals, Simulations, and Advanced Topics.* McGraw-Hill Publishing Company, May 1998.

[BM96] Costas Busch and Marios Mavronicolas. A combinatorial treatment of balancing networks. *Journal of the ACM*, 43(5):794–839, September 1996.

[DHW97] Cynthia Dwork, Maurice Herlihy, and Orli Waarts. Contention in shared memory algorithms. *Journal of the ACM*, 44(6):779–805, November 1997.

[DR01] P. Druschel and A. Rowstron. PAST: A large-scale, persistent peer-to-peer storage utility. In IEEE, editor, *Eighth IEEE Workshop on Hot Topics in Operating Systems (HotOS-VIII). May 20–23, 2001, Schloss Elmau, Germany*, pages 75–80, 1109 Spring Street, Suite 300, Silver Spring, MD 20910, USA, 2001. IEEE Computer Society Press.

[FH01] Fatourou and Herlihy. Adding networks. In *DISC: International Symposium on Distributed Computing*. LNCS, 2001.

[FLP85] M. J. Fischer, N. A. Lynch, and M. S. Paterson. Impossibility of distributed consensus with one faulty processor. *Journal of the ACM*, 32(2):374–382, 1985.

[Her91] M. Herlihy. Wait-free synchronization. *ACM Transactions on Programming Languages and Systems*, 11(1):124–149, January 1991.

[HR95] M. Herlihy and S. Rajsbaum. Algebraic topology and distributed computing — A primer. *Lecture Notes in Computer Science*, 1000:203–217, 1995.

[HS99] Maurice Herlihy and Nir Shavit. The topological structure of asynchronous computability. *Journal of the ACM*, 46(6):858–923, November 1999.

[HSW96] Maurice Herlihy, Nir Shavit, and Orli Waarts. Linearizable counting networks. *Distributed Computing*, 9(4):193–203, 1996.

[HW90] Maurice P. Herlihy and Jeannette M. Wing. Linearizability: A correctness condition for concurrent objects. *ACM Transactions on Programming Languages and Systems*, 12(3):463–492, July 1990.

[IL93] Amos Israeli and Ming Li. Bounded timestamps. *Distributed Computing*, 6(4):205–209, 1993.

[KBE+00] John Kubiatowicz, David Bindel, Patrick Eaton, Yan Chen, Dennis Geels, Ramakrishna Gummadi, Sean Rhea, Westley Weimer, Chris Wells, Hakim Weatherspoon, and Ben Zhao. OceanStore: An architecture for global-scale persistent storage. *ACM SIGPLAN Notices*, 35(11):190–201, November 2000.

[KLL+97] David Karger, Eric Lehman, Tom Leighton, Rina Panigrahy, Matthew Levine, and Daniel Lewin. Consistent hashing and random trees: distributed caching protocols for relieving hot spots on the World Wide Web. In ACM, editor, *Proceedings of the twenty-ninth annual ACM Symposium on the Theory of Computing: El Paso, Texas, May 4–6, 1997*, pages 654–663, New York, NY, USA, 1997. ACM Press.

[PRR97] C. Greg Plaxton, Rajmohan Rajaraman, and Andrea Richa. Accessing nearby copies of replicated objects in a distributed environment. In *9th Annual ACM Symposium on Parallel Algorithms and Architectures (SPAA '97)*, pages 311–320, New York, June 1997. Association for Computing Machinery.

[RFH+01] Sylvia Ratnasamy, Paul Francis, Mark Handley, Richard Karp, and Scott Shenker. A scalable Content-Addressable network. In Roch Guerin, editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 161–172, New York, August 27–31 2001. ACM Press.

[RRVV01] Rajmohan Rajaraman, Andréa W. Richa, Berthold Vöcking, and Gayathri Vuppuluri. A data tracking scheme for general networks. In *Proceedings of the 13th Annual ACM Symposium on Parallel Algorithms and Architectures*, pages 247–254, Crete Island, Greece, July 3–6, 2001. SIGACT/SIGARCH and EATCS.

[SMK+01] Ion Stoica, Robert Morris, David Karger, Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable Peer-To-Peer lookup service for internet applications. In Roch Guerin, editor, *Proceedings of the ACM SIGCOMM 2001 Conference (SIGCOMM-01)*, volume 31, 4 of *Computer Communication Review*, pages 149–160, New York, August 27–31 2001. ACM Press.

[VA86] P. M. B. Vitányi and B. Awerbuch. Atomic shared register access by asynchronous hardware. In *27th Annual Symposium on Foundations of Computer Science*, pages 233–243, Los Angeles, Ca., USA, October 1986. IEEE Computer Society Press.