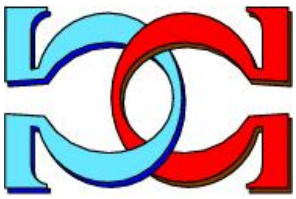

CDMTCS
Research
Report
Series

P Systems and the Byzantine Agreement

Michael J. Dinneen
Yun-Bum Kim
Radu Nicolescu

Department of Computer Science,
University of Auckland,
Auckland, New Zealand

CDMTCS-375
January 2010 (slightly revised April 2010)

Centre for Discrete Mathematics and
Theoretical Computer Science

P Systems and the Byzantine Agreement

Michael J. Dinneen, Yun-Bum Kim and Radu Nicolescu
Department of Computer Science, University of Auckland,
Private Bag 92019, Auckland, New Zealand

Abstract

We first propose a modular framework for recursive composition of P systems. This modular approach provides encapsulation and information hiding, facilitating the design of P programs for complex algorithms. Using this framework, we developed a P program that solves the classical version of the Byzantine agreement problem, for N participants connected in a complete graph, according to the well known Byzantine agreement algorithm based on EIG trees. We prove the correctness of this modular composition and conclude with a list of open problems.

1 Introduction

This paper continues our study [15, 14, 9] of P systems [17, 18] as modelling tools for distributed applications and networking, initially motivated by the investigations of Ciobanu *et al.* [8, 7]. We earlier proposed a new model for P systems, called *hyperdag P systems* [13, 15], in short *hP systems*, which allows more flexible communications than tree-based models, while preserving a strong hierarchical structure. To achieve our goals, this model has subsequently evolved [16, 14, 9] and it offers the following distinct facilities: (a) it extends the tree structure of classical P systems to *directed acyclic graphs* (dags); (b) it augments the operational rules of neural P systems (nP systems) [17] with *broadcast* facilities; (c) it refines the *rewriting* and *transfer* modes, associating these modes independently to each rule, instead of state; and (d) it allows the creation of *mobile* channels, which dynamically extend the structure of a considered P system model (analogous to nerves which extend in a regenerating tissue or threads extended by spiders). We have noticed that these adjustments, which enhance the model versatility, can also be retrofitted to other P system models.

Using this model, we developed basic building blocks in [14] that are relevant for network discovery (see also [11]): broadcast, convergecast, flooding, determine shortest paths and other basic metrics (such as, the number of nodes, descendants, paths).

We studied the well known *Firing Squad Synchronization Problem* (FSSP), in the framework of P systems [9]. We provided efficient solutions for the FSSP problem that have wider applicability than previous solutions [4, 2].

Here, we continue this study to address the possible existence of cells that are arbitrarily faulty. A well-studied problem in this area is known as the *Byzantine agreement*

problem, first proposed in 1980 [20]: reliable computer systems (or networks) must be able to handle malfunctioning components (or processes) that give conflicting information to different parts of the system. Lamport *et al.*'s description [10] is very readable and this problem has become one of the most studied problems in distributed computing—some even consider it the "crown jewel" of distributed computing. Lynch covers many versions of this problem and their solutions, including a complete description of the classical algorithm based on *Exponential Information Gathering* (EIG) trees [11].

Recent years have seen revived interest in this problem and its solutions, in a wide variety of contexts [6, 1, 5, 12], including, for example, solutions for quantum computers [3, 22]. To the best of our knowledge, no solution for P systems has been published. In the context of P systems, this problem was briefly mentioned, without solutions [8, 7]. We believe that we provide the first P systems solution for this problem. Our solution is based on the classical algorithm, using EIG trees.

In the course of this work, we realized that our framework was not versatile enough for a reasonable design. Following Paun *et al.*'s proposal [19], we propose a new modular framework, which supports encapsulation, information hiding and recursive composition. Our proposal is compatible with any data structure based on directed arcs, i.e. it covers cell-like P systems (based on trees), hP systems (based on dags) and nP systems (based on digraphs).

The rest of the paper is organized as follows. Section 2 covers a few basic preliminaries, then introduces our new modular framework, called P modules, and the recursive composition of P modules. We describe the Byzantine agreement problem in detail in Section 3, which also includes a small case study with four processes. Section 4 introduces the classical Byzantine agreement algorithm based on EIG trees. In Section 5, using our new modular framework, we model and develop the structure of a P systems implementation of the Byzantine agreement problem. The rules used in our design are described in Section 6. In Section 7, we prove the correctness of our modular design. Finally, in Section 8, we summarize our results and discuss related open problems.

2 Preliminaries

We assume that the reader is familiar with the basic terminology and notations: relations, graphs, nodes (vertices), arcs, directed graphs, dags, trees, alphabets, strings and multisets [13].

We first recall a few basic concepts from combinatorial enumerations. The *integer range* from m to n is denoted by $[m, n]$, i.e. $[m, n] = \{m, m + 1, \ldots, n\}$, if $m \leq n$, and $[m, n] = \emptyset$, if $m > n$. The set of *permutations* of n of length m is denoted by $P(n, m)$, i.e. $P(n, m) = \{\pi : [1, m] \to [1, n] \mid \pi \text{ is injective}\}$. A permutation π is represented by the sequence of its values, i.e. $\pi = (\pi_1, \pi_2, \ldots, \pi_m)$, and we will often abbreviate this further as the sequence $\pi = \pi_1.\pi_2 \ldots \pi_m$. The sole element of $P(n, 0)$ is denoted by $()$. Given a subrange $[p, q]$ of $[1, m]$, we define a *subpermutation* $\pi(p : q) \in P(n, q - p + 1)$ by $\pi(p : q) = (\pi_p, \pi_{p+1}, \ldots, \pi_q)$. The *image* of a permutation π, denoted by $Im(\pi)$, is the set of its values, i.e. $Im(\pi) = \{\pi_1, \pi_2, \ldots, \pi_m\}$. The *concatenation* of two permutations is denoted by $\oplus$, i.e. given $\pi \in P(n, m)$ and $\tau \in P(n, k)$, such that $Im(\pi) \cap Im(\tau) = \emptyset$,

$\pi \oplus \tau = (\pi_1, \pi_2, \ldots, \pi_m, \tau_1, \tau_2, \ldots, \tau_k) \in P(n, m+k)$.

The Byzantine agreement algorithm used later in this paper uses *Exponential Information Gathering* (EIG) trees as a data structure. An EIG tree $T_{N,L}$, $N \geq L \geq 1$, is a labelled (ordered) rooted tree of height L that is defined recursively as follows. The tree $T_{N,1}$ is a rooted tree with $1 + N$ nodes, with root labelled by λ and its N leaves labelled $1, 2, \ldots, N$, left to right. For $L > 1$, $T_{N,L}$ is a rooted tree with $1 + N|T_{N-1,L-1}|$ nodes (where $|T|$ is the size of tree T), root λ, having N subtrees, where each subtree is isomorphic with $T_{N-1,L-1}$ and each subtree node is labelled by the least element of $[1, N]$ that is different from any ancestor node or any left sibling node. Thus, there is a bijective correspondence between the permutations of $P(N, L)$ and the sequences (concatenations) of labels on all root-to-leaf paths of $T_{N,L}$. See Figure 2 for an example of the EIG tree $T_{4,2}$.

We also assume familiarity with P systems [17, 18], nP systems [18] or hP systems [15]. Although the P systems considered here can be described in these classical frameworks, we prefer to present them in a modular way, using a new definition, that subsumes their essential features and provides facilities for recursive modular composition.

Definition 1 (P module)**.** A *P module* is a system $\Pi = (O, K, \delta, P)$, where:

1. O is a finite non-empty alphabet of *objects*;
2. K is a finite set of *cells*, where each cell, $\sigma \in K$, has the form $\sigma = (Q, s_0, w_0, R)$, where:
   - Q is a finite set of *states*;
   - $s_0 \in Q$ is the *initial state*;
   - $w_0 \in O^*$ is the *initial multiset* of objects;
   - R is a finite *ordered* set of multiset rewriting *rules* of the general form: $s\ x \rightarrow_\alpha s'\ x'\ (u)_{\beta_\gamma}$, where $s, s' \in Q$, $x, x' \in O^*$, $u \in O^*$, $\alpha \in \{min, max\}$, $\beta \in \{\uparrow, \downarrow, \updownarrow\}$, $\gamma \in \{one, spread, repl\} \cup K$. If $u = \lambda$, this rule can be abbreviated as $s\ x \rightarrow_\alpha s'\ x'$. The meaning of operators α, β, γ is described at the end of this definition.
3. δ is a binary relation on K, i.e. a set of parent-child structural arcs, representing *duplex* or *simplex* communication channels between cells;
4. P is a subset of K, indicating the *port* cells, i.e. the only cells can be connected to other modules.

The rules given by the ordered set R are attempted in *weak priority* order [18]. If a rule is *applicable*, then it is *applied* and then the next rule is attempted (if any). If a rule is not applicable, then the next rule is attempted (if any). Note that state-based rules introduce an extra requirement for determining rule applicability, namely the target state indicated on the right-hand side must be the same as the previously chosen target state (if any) [17, 13, 15]. Rules are applied under the usual eager evaluation of their left-hand sides and lazy evaluation of their right-hand sides [17].

With these conventions, one cell's ordered set of rules becomes a sequence of programming statements for a hypothetical P machine, where each rule includes a simple if-then-fi conditional test for applicability and, as we see below, some while-do-od looping facilities (*max* and *repl* operators), with some potential for in-cell parallelism, in addition to the more obvious inter-cell parallelism. State compatibility introduces another intra-cell if-then-fi conditional test, this time between rules.

The *rewriting* operator $\alpha = min$ indicates that the rewriting is applied once, if the rule is applicable; and $\alpha = max$ indicates that the rewriting is applied as many times as possible, if the rule is applicable. Here, we intentionally avoid the $\alpha = par$ operator, because we do not use it and it is more complicated to integrate it into a priority scheme.

The *transfer* operator $\beta = \uparrow$ indicates that the multiset u is sent "up" to the parents; $\beta = \downarrow$ indicates that the multiset u is sent "down" to the children; and $\beta = \updownarrow$ indicates that the multiset u is sent both "up" and "down". For simplicity, here we intentionally avoid other operators that we do not use in this paper, such as $\beta = \leftrightarrow$, which indicates transfer to the siblings.

The additional transfer operator $\gamma = one$ indicates that the multiset u is sent to one recipient (parent or child, according to the direction indicated by β). The operator $\gamma = spread$ indicates that the multiset u is spread among an arbitrary number of recipients (parents, children or parents and children, according to the direction indicated by β). The operator $\gamma = repl$ indicates that the multiset u is replicated and broadcast to all recipients (parents, children or parents and children, according to the direction indicated by β). The operator $\gamma = \sigma \in K$ indicates that the multiset u is sent to σ, if cell σ is in the direction indicated by β; otherwise, the multiset u is "lost". By convention, if cells have unique indices or are labelled and labels are locally unique, we can abbreviate $\gamma = \sigma$ by $\gamma = i$, where i is the index or label of σ.

The following examples illustrate the behavior of these operators. Consider a cell σ, in state s and containing aa. Consider the potential application of a rule $s\ a \rightarrow_\alpha s'\ b\ (c)_{\beta_\gamma}$, by looking at specific values for α, β, γ operators:

- The rule $s\ a \rightarrow_{min} s'\ b\ (c)_{\uparrow_{repl}}$ can be applied and, after its application, cell σ will contain ab and a copy of c will be sent to each of σ's parents.

- The rule $s\ a \rightarrow_{max} s'\ b\ (c)_{\uparrow_{repl}}$ can be applied and, after being applied twice, cell σ will contain bb and a copy of cc will be sent to each of σ's parents.

- The rule $s\ a \rightarrow_{min} s'\ b\ (c)_{\downarrow_{\sigma'}}$ (where $\sigma' \in K$), can be applied and, after its application, cell σ will contain ab and a copy of c will be sent to σ', if σ' appears among the children of σ, otherwise, this c will be lost.

- The rule $s\ a \rightarrow_{max} s'\ b\ (c)_{\downarrow_{\sigma'}}$ (where $\sigma' \in K$) can be applied and, after being applied twice, cell σ will contain bb and a copy of cc will be sent to σ', if σ' appears among the children of σ, otherwise, this cc will be lost.

In this paper, we are only interested in *deterministic solutions*, and we will exclusively use the *min*, *max*, *repl*, and K operators, and avoid operators with a higher potential for non-determinism, such as *par*, *one*, *spread*.

By default, unless specifically mentioned, the channels are *duplex*, allowing simultaneous transmissions from both ends. *Simplex* channels are explicitly specified, and indicate a single open direction, either from parent to child, or from child to parent (there is no necessary relation between the structural directions and communication direction); messages sent in the other direction are "lost".

This definition of P module subsumes several earlier definitions of P systems, hP systems and nP systems. If δ is a *tree*, then P is essentially a tree-based P system (which can also be interpreted as a cell-like P system). If δ is a *dag*, then P is essentially an hP system. If δ is a *digraph*, then P is essentially an nP system.

Given an arbitrary finite set of P modules, we can construct a higher level P module by creating channels between ports of the given P modules. This construction requires that the original P modules have disjoint cells.

Consider a finite family of n P modules, $\mathcal{P} = \{\Pi_i \mid i \in [1,n]\}$, where $\Pi_i = (O_i, K_i, \delta_i, P_i)$, $i \in [1,n]$. This family $\mathcal{P}$ is *cell-disjoint*, if their cell sets are disjoint, i.e. $K_i \cap K_j = \emptyset$, for $i, j \in [1,n]$. If required, any such family can be made cell-disjoint, by a *deep copy* process, which clones all cells and, as a convenience, automatically allocates successive indices to cloned cells (e.g., starting from cell σ, the first cloned cell is σ_1, the second is σ_2, etc).

Definition 2 (P module composition)**.** The P module $\Psi = (O, K, \delta, P)$ is a *composition* of the P module family $\mathcal{P}$, if:

- $\mathcal{P}$ is cell-disjoint,
- $O = \bigcup_{i\in[1,n]} O_i$,
- $K = \bigcup_{i\in[1,n]} K_i$,
- $\delta = \delta' \cup \bigcup_{i\in[1,n]} \delta_i$, where δ' is a binary relation on $\bigcup_{i\in[1,n]} P_i$,
- $P \subseteq \bigcup_{i\in[1,n]} P_i$.

In this case, the P modules in $\mathcal{P}$ are called *components* of Ψ.

When defining a new P module composition, we only need two items: (1) the additional δ' relation and (2) the remaining port set P. To simplify the discourse, we will use this approach, and omit the description of the other components, which are always the same in any P module composition.

This modular approach provides encapsulation, information hiding and recursive composition, facilitating the design of P programs for complex algorithms.

The following definition embodies the idea of a P system with rules which depend on generic objects, which can be specified at a later stage.

Definition 3 (Generic P module)**.** A *generic P module* is a system $\Pi\langle x_1, x_2, \ldots, x_n\rangle = (O, K, \delta, P)$, where its generic parameters, $x_1, x_2, \ldots, x_n$, designate fixed, but yet unspecified, objects. These generic parameters can be used as additional objects in the definition of its rules and must be instantiated to actual objects in O, before the rules can be applied. For each cell, its rule sequence is also generic on $\langle x_1, x_2, \ldots, x_n\rangle$, emphasizing that these additional symbols can be used as objects.

Instantiation is indicated by assigning specific objects to generic parameter names, and is accomplished by an automatic *deep copy* plus a *textual substitution* of the generic parameter names by their associated specific objects. We accept both *total instantiations*, which fix all generic parameters of a generic P module, and *partial instantiations*, which only fix a subset of the generic parameters.

For example, consider a generic P module $\Pi\langle x, y\rangle = (O, \{\sigma_{xy}, \tau_{xy}\}, \{\sigma_{xy} \rightarrow \tau_{xy}\}, \{\sigma_{xy}\})$, where x and y are its generic parameters, and assume that the object set O includes the digits. Then, $\Pi\langle x = 2, y = 3\rangle = (O, \{\sigma_{23}, \tau_{23}\}, \{\sigma_{23} \rightarrow \tau_{23}\}, \{\sigma_{23}\})$ is a total instantiation, and $\Pi\langle x, y = 3\rangle = (O, \{\sigma_{x3}, \tau_{x3}\}, \{\sigma_{x3} \rightarrow \tau_{x3}\}, \{\sigma_{x3}\})$ is a partial instantiation.

As suggested above, a good practice is to systematically index all cells of a generic P module, by the names of the generic parameters or of their actual associated objects (or their indices). Although not required, we will generally follow this convention.

Composing generic P modules, or a mixture of generic and non-generic P modules, constructs another generic P module. Depending on the intended effect, generic parameter names can be freely renamed (or not), as needed. For example, we can combine $\Pi\langle x = 3, y\rangle$, $\Pi\langle x = 4, y\rangle$ into a generic P module $\Gamma\langle b\rangle$; and $\Pi\langle x = 3, y_1\rangle$, $\Pi\langle x = 4, y_2\rangle$ into a generic P module $\Delta\langle y_1, y_2\rangle$.

While generic P modules are not strictly needed, we will use them to better manage the design complexity.

3 Byzantine agreement problem

We first introduce an anthropomorphic version of the Byzantine agreement problem:

> The Byzantine Generals' Problem is an agreement problem (first proposed by Pease *et al.* [20]) in which N generals of the Byzantine Empire's army must unanimously decide whether to attack some enemy army or to retreat.
>
> The problem is complicated by the geographic separation of the generals, who must communicate by sending messengers to each other, and by the possible presence of up to F traitors amongst the N generals. These traitors can act arbitrarily in order to achieve the following aims: trick some generals into attacking; force a decision that is not consistent with the generals' desires, e.g., forcing an attack when no general wished to attack; or confusing some generals to the point that they are unable to make up their minds. If the traitors succeed in any of these goals, any resulting attack is doomed, as only a concerted effort can result in victory.
>
> Byzantine fault tolerance can be achieved if the loyal (non-faulty) generals have a unanimous agreement on their strategy. Note that if all loyal generals start with the same initial assessment value, attack or retreat, they must in the end agree upon the same value. Otherwise, the choice of strategy agreed upon is irrelevant.
>
> The Byzantine failure assumption models real-world environments in which computers and networks may behave in unexpected ways due to hardware

> failures, network congestion and disconnection, as well as malicious attacks. Byzantine failure-tolerant algorithms must cope with such failures and still satisfy the specifications of the problems they are designed to solve. Such algorithms are commonly characterized by their resilience F, the number of faulty processes with which an algorithm can cope.
>
> [Taken from [21], with minor changes]

In less anthropomorphic terms, we are given N distributed participants (or processes), which model the N generals. A small number F of these participants model the traitors and are called Byzantine faulty, i.e. they can fail in any possible way. All the faults considered here are Byzantine faults and we will use the abbreviations *fault* and *faulty* to mean Byzantine fault and Byzantine faulty, respectively. Example of faults are: sending different messages to different participants, sending incorrect messages, refraining from sending messages. Briefly, it can do anything that has a chance of disrupting the agreement. The other $N - F$ participants model the loyal generals and are called correct. Correct participants never fail and strictly follow the same agreement algorithm. The initial decision values are conventionally represented by a single bit, e.g., 1 for "attack" and 0 for "retreat".

In this paper, we consider only the basic scenario, where each pair of participants is connected by a fully reliable duplex channel and the resulting network works synchronously. It is well known that, in this basic case, the agreement is possible, if and only if $N \geq 3F + 1$. Note that, outside this basic scenario, the agreement is still possible for $2F + 1$ connected communication graphs, for channels with specific bounds on faults, for asynchronous networks with specific bounds on delays. However, agreement is not possible for arbitrary communication graphs, for arbitrary communication faults or for unbounded delays. These topics are not further discussed here and, for further details, refer to Lynch [11].

Figure 1 illustrates this basic case. We have four participants, P_1, P_2, P_3, P_4, with initial values 0, 0, 1, 1, respectively. These four participants are connected in a complete graph, often with loopback arcs (useful for uniform treatment), where the arcs indicate fully reliable channels. We assume that P_2, P_3, P_4 are correct, but P_1 is faulty. In this case, correct participants can agree, because $N = 4$, $F = 1$, and $N \geq 3F + 1$. Each participant has its own exact copy of one of the existing algorithms which solves the Byzantine agreement problem. In the next section, we review the first classical algorithm for this problem and illustrate how the agreement is always reached, despite the effort of the faulty participant P_1.

4 Classical Byzantine agreement algorithm based on EIG trees

The classical solution of the Byzantine agreement problem uses EIG $T_{N,L}$ trees as a data structure, and guarantees a solution if $N \geq 3F + 1$ and $L = F + 1$. It also uses a built-in default value, W (called V_0 [11]), to break ties and to replace wrong or missing messages. These parameters N, L and W are global and "hardcoded" into its rules.

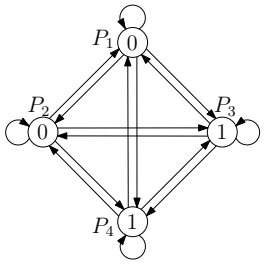

Figure 1: A Byzantine agreement problem, with $N = 4$.

A complete description of this algorithm is available in Lynch [11]. We give a simplified description, illustrating this on a particular case, where $N = 4$, $F = 1$, $L = 2$, $W = 0$. We assume that participants P_1, P_2, P_3, P_4 start with the initial values 0, 0, 1, 1, respectively, as shown in Figure 1. Participants P_2, P_3, P_4 are assumed correct, but P_1 is faulty and therefore allowed to send out arbitrary messages, if it chooses so.

This algorithm works in two distinct phases: first, a messaging phase, where the EIG trees are populated with the received messages, in a top-down order and, secondly, an evaluation phase which works bottom-up on the EIG trees.

We use apostrophes ($'$) and quotation marks ($''$) to mark top-down values and bottom-up values, respectively (Lynch designates these values *val* and *newval* [11]).

4.1 Phase I: messaging and filling top-down values

Phase I consists of L messaging rounds, which fill the EIG trees, top-down, one tree level per round. In our specific example, there will be two messaging rounds, respectively filling the first and the second EIG tree levels.

In the first messaging round, each correct participant, P_i, sends a copy of its initial decision value, v_i, to each of all four participants, i.e. to the other three participants and, using a loopback interface, back to itself, $P_i \overset{v_i}{\Rightarrow} P_j$, $j \in [1, 4]$. For example, the correct participant P_2, with initial value $v_0 = 0$, sends out four identical 0 messages to all four participants: $P_2 \overset{0}{\Rightarrow} P_i$, $i \in [1, 4]$. The other two correct participants, P_3 and P_4, proceed similarly: $P_3 \overset{1}{\Rightarrow} P_i$, $i \in [1, 4]$, $P_4 \overset{1}{\Rightarrow} P_i$, $i \in [1, 4]$.

A correct participant P_1 would have also been expected to send out identical messages to all participants, according to its initial decision value, 0 in our example. However, P_1 is faulty and can send out conflicting messages, if it wishes so. For example, P_1 sends 0 to each of P_1, P_2, P_3, but 1 to P_4: $P_1 \overset{0}{\Rightarrow} P_i$, $i \in [1, 3]$, $P_1 \overset{1}{\Rightarrow} P_4$.

As the channels are all reliable, all messages are properly received. Participant P_2 receives the following four messages: $P_2 \overset{0}{\Leftarrow} P_1$, $P_2 \overset{0}{\Leftarrow} P_2$, $P_2 \overset{1}{\Leftarrow} P_3$, $P_2 \overset{1}{\Leftarrow} P_4$. Participants P_1 and P_3 receive the same messages as P_2, for example: $P_3 \overset{0}{\Leftarrow} P_1$, $P_3 \overset{0}{\Leftarrow} P_2$, $P_3 \overset{1}{\Leftarrow} P_3$, $P_3 \overset{1}{\Leftarrow} P_4$. However, one of the messages received by P_4 differs: $P_4 \overset{1}{\Leftarrow} P_1$, $P_4 \overset{0}{\Leftarrow} P_2$, $P_4 \overset{1}{\Leftarrow} P_3$, $P_4 \overset{1}{\Leftarrow} P_4$.

All round 1 received values are now stored in the EIG trees, in a position related to the sender's identity, which is known by the receiver. Each correct participant P_k uses its EIG node i to store the value v received from participant P_i. For example, P_2 stores

$0'$, $0'$, $1'$ and $1'$, in its EIG nodes 1, 2, 3 and 4, respectively, as also shown by level 1 EIG nodes of Figure 2. Level 1 EIG nodes of Figures 3 and 4 illustrate the round 1 messages received and stored by P_3 and P_4, respectively.

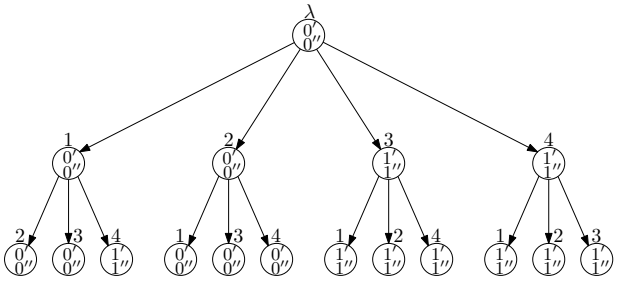

Figure 2: EIG tree for P_2.

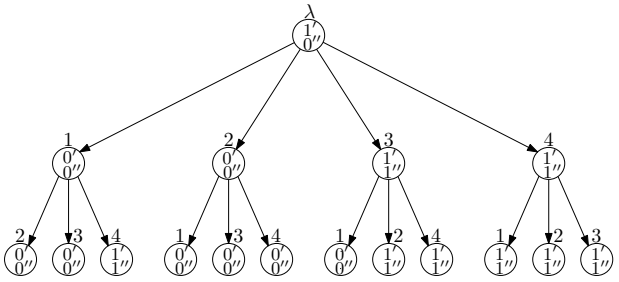

Figure 3: EIG tree for P_3.

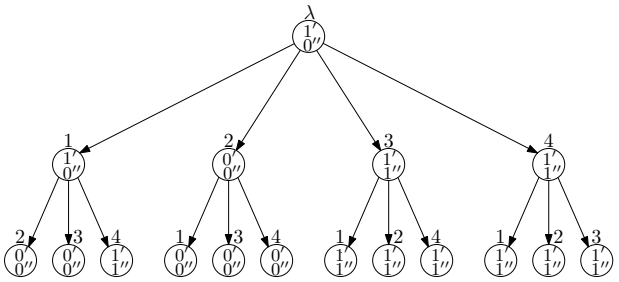

Figure 4: EIG tree for P_4.

In the second messaging round, each correct participant further relays copies of round 1 received messages, to all four participants. All received messages are faithfully relayed, except where this would create loops. For this round, this means that a participant will not relay messages that have been received via its own loopback interface (i.e. a message originated from itself at round 1). For example, participant P_2 will not further relay the message 0 received from itself: $P_2 \overset{0}{\Rightarrow} P_2$.

The messages are sent using a protocol that identifies the original source of each message. Although not size optimal, we will here assume a straightforward protocol, which prefixes each message with the ID of the originator. For example, P_2 sends out

four identical messages to all four participants: $P_2 \overset{(1,0)(3,1)(4,1)}{\Longrightarrow} P_i$, $i \in [1,4]$. The other two correct participants, P_3 and P_4, proceed in a similar way: $P_3 \overset{(1,0)(2,0)(4,1)}{\Longrightarrow} P_i$, $i \in [1,4]$, $P_4 \overset{(1,1)(2,0)(3,1)}{\Longrightarrow} P_i$, $i \in [1,4]$.

Again, our faulty participant P_1 can, if it wishes, send out conflicting messages, which may or may not be consistent with its received values. For example, consider that P_1 sends out the following round 2 messages: $P_1 \overset{(2,0)(3,0)(4,1)}{\Longrightarrow} P_3$, $P_1 \overset{(2,0)(3,1)(4,1)}{\Longrightarrow} P_i$, $i \in \{1,2,4\}$.

All messages are properly received and stored in level 2 EIG nodes. Each correct participant P_k uses its EIG node $i.j$ to store the value v received from P_j via the message $P_j \overset{(i,v)}{\Longrightarrow} P_k$. For example, participant P_2 stores the values $0'$, $0'$, $0'$, $0'$, $0'$, $0'$, $1'$, $1'$, $1'$, $1'$, $1'$, $1'$, in its EIG nodes 1.2, 1.3, 1.4, 2.1, 2.3, 2.4, 3.1, 3.2, 3.4, 4.1, 4.2, 4.3, respectively. Level 2 EIG nodes of Figures 2, 3 and 4 illustrate the round 2 messages received and stored by participants P_2, P_3 and P_4.

In our case, the messaging rounds end after filling two levels in the EIG trees. However, in general, messaging will continue, using a similar mechanism, until all EIG levels are completely filled. Essentially, each correct participant P_k will use its EIG node $i_1.i_2 \ldots i_t.i_{t+1}$ to store the value v received from $P_{i_{t+1}}$ via the message $P_{i_{t+1}} \overset{(i_1.i_2 \ldots i_t, v)}{\Longrightarrow} P_k$. A correct participant $P_{i_{t+1}}$ will forward such a message only if it does not create a loop, i.e. if $i_{t+1} \notin \{i_1, i_2, \ldots, i_t\}$. The recursive application of this loop avoiding strategy ensures that the sequence $i_1.i_2 \ldots i_t.i_{t+1}$ is one of the permutations of $[1, N]$ of size $t+1$ and the value v always finds its proper unique place in the EIG tree.

Intuitively, this v is claimed to be the initial value of P_{i_1}, further relayed to P_k via P_{i_2}, $\ldots$, P_{i_t}, $P_{i_{t+1}}$, in this order. In fact, this is indeed the case, if all these participants are correct. For further and more precise details, see Lynch [11].

4.2 Phase II: evaluating bottom-up values

Phase II consists of L evaluation rounds, which proceed level by level, in a bottom-up manner.

First, for a leaf EIG node, the bottom-up value is set equal to its already filled top-down value. In our example, for participant P_2, the EIG nodes 1.2, 1.3, 1.4, 2.1, 2.3, 2.4, 3.1, 3.2, 3.4, 4.1, 4.2, 4.3, evaluate the following bottom-up values: $0''$, $0''$, $0''$, $0''$, $0''$, $0''$, $1''$, $1''$, $1''$, $1''$, $1''$, $1''$, respectively.

Next, assume that the bottom-up values have already been evaluated for level $L-t$, $t \in [0, L-1]$. The bottom-up values at the next higher level, $L-t-1$, are evaluated using a strict majority rule, or, if there is no strict majority, the result is the default value W (0 in our case). For example, using the strict majority rule, participant P_2's EIG nodes 1, 2, 3, 4, evaluate the bottom-up values $0''$, $0''$, $1''$, $1''$, respectively. However, at the next round, no strict majority exists at the EIG node λ. This tie is broken using the default value $0''$. Because λ is the EIG root, the final value for P_2 is 0.

Figures 2, 3 and 4 illustrate all bottom-up values evaluated by participants P_2, P_3 and P_4, respectively. We can see that all correct participants reach a common final decision. Although this is not required by the formal specifications of the Byzantine agreement problem, this common decision can also be reached by the faulty participant

P_1, regardless of its arbitrary outgoing messages, if it bothers to properly fill and evaluate an EIG tree. For further and more precise details, again see Lynch [11].

This brief example illustrates some fundamental properties of the EIG-based Byzantine agreement algorithm. The correct participants always reach a common decision, as long as the number of faulty participants does not exceed the prescribed bound F (here $F = 1$). In some border cases, by "cleverly" sending out inconsistent messages, the faulty participants are able to sway the common decision one way or another, but never to disrupt it.

We can show this by reconsidering the above example, with the only difference that, at the first messaging round, the faulty participant P_1 sends out 1 (instead of 0) to P_3, $P_1 \stackrel{1}{\Rightarrow} P_3$. In this case, all correct participants, P_2, P_3, P_4, will all reach the final decision 1 (instead of 0). They will still agree on a common decision value.

5 P system program for the Byzantine agreement

The following global parameters are known in advance and "hard-coded" into our current model: N, the number of participants, L, the height of the EIG trees, and W, the default value, for wrong or missing values.

We design our program by recursive composition of simpler P modules. The common vocabulary, O, used by all P modules includes the set $\{v, v', v'' \mid v \in \{0,1\}\} \cup \{?, *\} \cup \{x^v_\pi \mid v \in \{0,1,?\}, \pi \in P(N,t), t \in [0,L]\}$. Depending on the objects sent by faulty P modules, O can be larger than this set, as we cannot constrain the behavior of faulty participants in any way.

Objects 0 and 1 designate decision values. Objects $0'$ and $1'$ represent decision values stored as top-down values in the EIG trees. Objects $0''$ and $1''$ represent decision values stored as bottom-up values in the EIG trees. Object ? is a template that can match any decision value, 0 or 1. Object $*$ designates the last step in the top-down evaluation.

The object $x^v_{i_1.i_2\ldots i_t}$ represents a t^{th} round message received from P_{i_t}, i.e. using our earlier notation, $x^v_{i_1.i_2\ldots i_t} = (i_1.i_2\ldots i_t, v)$, where the right-hand side string is considered a single object. To simplify the notations, we also use the following natural conventions: $x^v_{i_1.i_2\ldots i_t} = x^v_{i_1 i_2\ldots i_t}$ and $x^v_{()} = x^v = v$. These notations are summarized in Figure 5. We prefer the x^v_π notation when we want to emphasize an "atomic" vocabulary object, and the (π, v) notation when we work on its constituent "sub-atomic" objects.

$$\begin{array}{lcl} x^v & = & v, \text{ for } v \in \{0,1\} \\ x^v_\pi & = & (\pi, v), \text{ for } v \in \{0,1\}, \\ & & t \in [0,L],\ \pi \in P(N,t) \end{array}$$

Figure 5: Notations summary (left, "atomic" notation; right, "sub-atomic" insight).

Our design uses the following elementary P modules: Ψ, a P module representing the "core" of a participant in a Byzantine decision; Θ, a P module representing an

EIG tree; and $\Gamma\langle h, f\rangle$, a generic P module, which takes care of all communications of participant h, with another participant f, if $h \neq f$, or with self, otherwise. Using these elementary P modules, as basic building blocks, we compose the generic P module $\Pi\langle h\rangle$, which represents a correct participant with index h, and, finally, the P module Ω, which represents a complete Byzantine scenario.

Figure 6 illustrates the generic P module $\Pi\langle h\rangle$, for the case $N = 4$ and $L = 2$, including its constituent P modules: Ψ, Θ, $\Gamma\langle h, f = 1\rangle$, $\Gamma\langle h, f = 2\rangle$, $\Gamma\langle h, f = 3\rangle$, $\Gamma\langle h, f = 4\rangle$. Dotted lines represent P module borders and shaded cells are the remaining ports of the figure's top P module, $\Pi\langle h\rangle$. As this figure clearly shows, Θ is (as expected) based on a *tree*, which is further included in the *dag* underlying the participant $\Pi\langle h\rangle$. The rest of this section clarifies this construction. We will first focus on the structural details and consider the rules after these are completed.

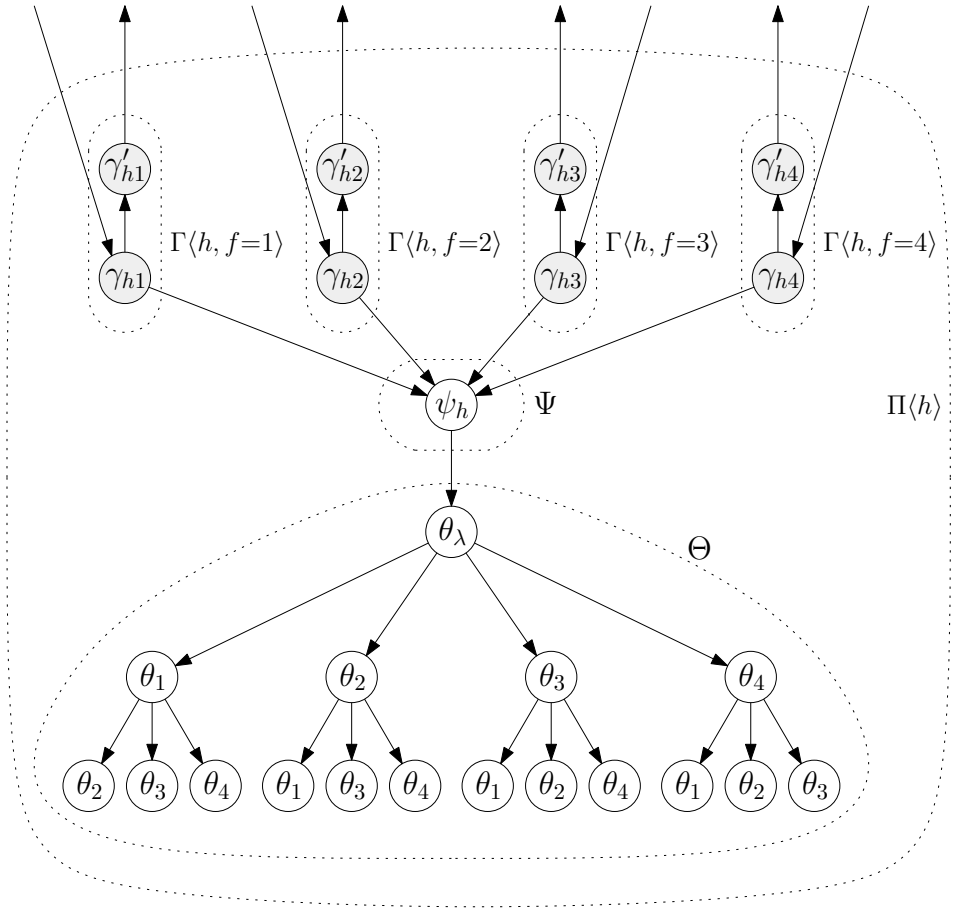

Figure 6: The P module $\Pi\langle h\rangle$, for $N = 4$, $L = 2$.

The P module Ψ contains a single cell and is defined by: $\Psi = (O, K_\Psi, \emptyset, P_\Psi)$, where $K_\Psi = P_\Psi = \{\psi\}$, $\psi = (Q_\psi, s_0, v, R_\psi)$, $Q_\psi = \{s_i \mid i \in [0, L]\} \cup \{s_{L+1}\}$, $v \in \{0, 1\}$ is the initial decision value of this participant and the rule sequence R_Ψ is given in the next section. The P module Ψ does not need to be generic, its rules are identical for all participants.

The P module Θ contains the EIG tree and is defined by: $\Theta = (O, K_\Theta, \delta_\Theta, P_\Theta)$.

Essentially, K_Θ and δ_Θ define an EIG tree as previously described, for the global parameters N and L. The root cell of the tree is labelled θ_λ, which is also its single port, $P_\Theta = \{\theta_\lambda\}$. All EIG cells start with empty contents and share the same states, $Q_\Theta = \{d_t \mid t \in [0, L]\} \cup \{u_t \mid t \in [0, 5]\} \cup \{t_z\}$, and rule sequence R_Θ, which is given in the next section. The P module Θ does not need to be generic, its rules are identical for all participants.

The generic P module $\Gamma\langle h, f\rangle$ contains two cells and is defined by: $\Gamma\langle h, f\rangle = (O, K_\Gamma, \delta_\Gamma, P_\Gamma)$, where $K_\Gamma = P_\Gamma = \{\gamma_{hf}, \gamma'_{hf}\}$, $\delta_\Gamma = \{\gamma_{hf} \to \gamma'_{hf}\}$, $\gamma_{hf} = (Q_\gamma, p_0, \emptyset, R_\gamma\langle h, f\rangle)$, $Q_\gamma = \{p_t, q_t, r_t \mid t \in [0, L-1]\} \cup \{p_L, p_{L+1}\}$, $\gamma'_{hf} = (Q'_\gamma, c_0, \emptyset, R'_\gamma)$, $Q'_\gamma = \{c_t \mid t \in [0, 3]\}$.

The rule sequences $R_\gamma\langle h, f\rangle$ (generic) and R'_γ (non-generic) are given in the next section. After constructing the higher levels P modules (Π and Ω), these generic parameters will be fixed: $h \in [1, N]$, as the index of the participant which contains it (its "home"); and $f \in [1, N]$, as the index of the participant at the other connection end (a potentially faulty "friend-or-foe"). Although not strictly necessary, this generic approach facilitates a uniform design.

We now design a higher generic P module, $\Pi\langle h\rangle = (O, K_\Pi, \delta_\Pi, P_\Pi)$, representing a generic Byzantine participant, with index h, by composing: one deep copy of P module Ψ (the main cell), one deep copy of P module Θ (the EIG tree), and the following N partial instances of the generic P module $\Gamma\langle h, f\rangle$: $\Gamma\langle h, f = 1\rangle$, $\Gamma\langle h, f = 2\rangle$, $\ldots$, $\Gamma\langle h, f = N\rangle$. To complete the definition, we define its additional arcs, $\delta'_\Pi = \{\psi_h \to \theta_\lambda\} \cup \{\gamma_{hj} \to \psi_h \mid j \in [1, N]\}$, and its remaining port set, $P_\Pi = \{\gamma_{hj} \mid j \in [1, N]\}$.

It might be useful, at this stage, to have a second look at Figure 6. In this case, $N = 4$ and the P module $\Pi\langle h\rangle$ has four groups of two ports available for further connections—one group for each participant (including self). Single ended arrows indicate how this participant will be finally connected.

To complete the design, we define the final composition, $\Omega = (O, K_\Omega, \delta_\Omega, P_\Omega)$, representing a complete Byzantine scenario, by composing the following N instances of the P module $\Pi\langle h\rangle$: $\Pi\langle h = 1\rangle$, $\Pi\langle h = 2\rangle$, $\ldots$, $\Pi\langle h = N\rangle$. We define its remaining port set, $P_\Omega = \emptyset$ (assuming that Ω does not need to be further connected); and its additional arcs, $\delta'_\Omega = \{\gamma'_{ij} \to \gamma_{ji} \mid i, j \in [1, N]\}$, where these new structural arcs work as *simplex* communication channels, only from transmission from child (γ_{ij}) to parent (γ'_{ij}). At this stage, we have completed the customization of parameters f and h, for each constituent $\Gamma\langle i, j\rangle$, $i, j \in [1, N]$.

Figure 7 illustrates a fragment of the P module Ω (for $N = 4$ and $L = 2$) showing the channels between two participants, Π_2 and Π_3; all other connection pairs are similar. As expected, the last added channels define a complete graph among the participants, where each participant has also a *loopback* connection, corresponding to the channels $\gamma'_{ii} \to \gamma_{ii}$, $i \in [1, N]$.

To start the system, we "magically" drop the initial decision values in participants' main cells, ψ_i, for $i \in [1, N]$. Thereafter, all cells start in the same time step, synchronously, as in the standard Byzantine agreement algorithms [11].

Remark 4. The number of cells in the P module Π grows exponentially with N, the number of participants, and L, the height of the EIG trees used. It is easy to see by induction, that each EIG tree $T_{N,L}$ contains at most $2(N)_L$ nodes, where $(N)_L$ denotes

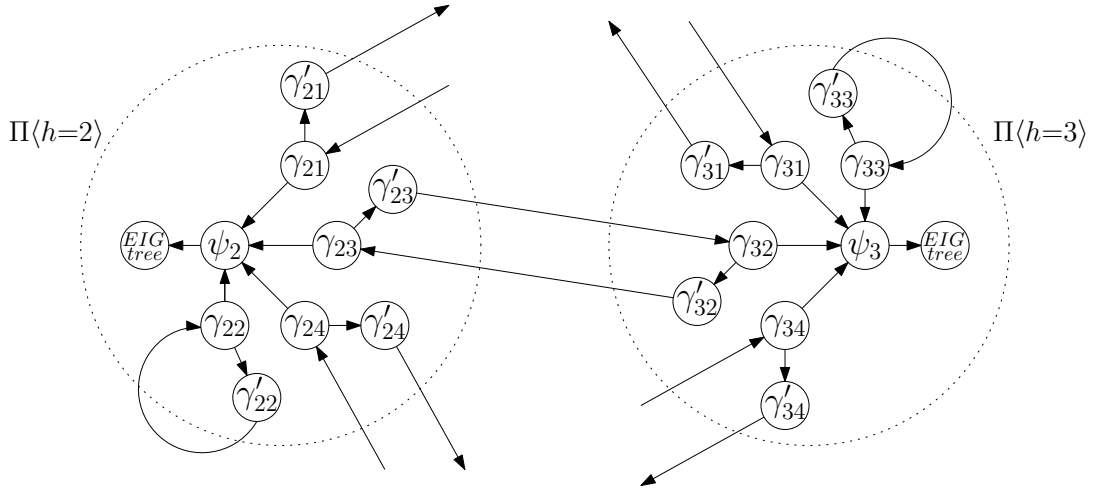

Figure 7: Fragment of the P module Ω, for $N = 4$, $L = 2$, showing connections between participants 2 and 3 (here, node contents indicate cell indices).

the falling factorial $N(N-1)\cdots(N-L-1)$. Thus, since our P module construction needs N copies, we have an upper bound of $O((N)_L N)$ number of cells.

As mentioned earlier, in any Byzantine agreement algorithm, the maximum number of tolerated faults is $F = (N-1)/3$. Also, in the EIG algorithm, the maximum number of tolerated faults is bounded by the height of the EIG tree, $F \leq L-1$. Therefore, for maximum fault tolerance, L is linearly related to N, $L = (N-1)/3+1$, and therefore we conclude that the number of cells in Π grows exponentially with N.

6 Byzantine agreement rules

This section lists the four rule sequences which appear in our modules' definition: $R_\gamma\langle h, f\rangle$ and R'_γ for module $\Gamma\langle h, f\rangle$, R_Ψ for module Ψ, and R_Θ for module Θ. Rulesets $R_\gamma\langle h, f\rangle$, R'_γ, R_Ψ and part of R_Θ (states d_0 to d_L) simulate Phase I of the classical EIG-based algorithm (described in Section 4.1), the messaging and the filling of top-down values. The remaining R_Θ rules (states u_0 to u_6) simulate the bottom-up evaluation Phase II.

We summarize these rule sequences as templates, which (most of them) depend on the "hard-coded" global parameters N, L and W. As earlier mentioned, several rules for $R_\gamma\langle h, f\rangle$ depend additionally on the generic objects h ("home") and f ("friend-or-foe"). Note that, for some rule templates, the number of corresponding actual rules grows exponentially with N and L.

To facilitate the understanding of our rules, each rule sequence is preceded by a statechart, graphically illustrating state transitions. Where several rules are grouped together, their relative order is omitted as irrelevant, because they start from the same state, end in the same state, and their left-hand sides are disjoint. Also, each rule template, which refers to permutations of size L or $L-1$, is followed by an itemized expansion for the sample case $L = 2$ (as used by our specific examples).

6.1 Rule sequences $R_\gamma\langle h, f\rangle$ and R'_γ

These rule sequences define the behavior of cells in module $\Gamma\langle h, f\rangle$. The ruleset $R_\gamma\langle h, f\rangle$ is for cells γ_{hf} and the ruleset R'_γ for cells γ'_{hf}. The state transitions for these sequences are illustrated in Figure 8. State p_L is the final state for cell γ_{hf}, i.e. no further transition is possible from this state. Cell γ'_{hf} does not have a distinct final state; however, cell γ'_{hf} will normally end in state c_2. For ruleset (4), a message sent up by *repl* is an internal message sent to cell γ_{hf}. For rulesets (8, 9), a message sent up by *repl* is an *external* message sent to cell γ'_{fh} (i.e. to another participant). For rulesets (11, 12, 13), a message sent down by *repl* is an internal message sent to the main cell ψ_h (a copy of it is also sent to γ'_{hf}, where it is discarded). These rules are further discussed in Section 7.1.

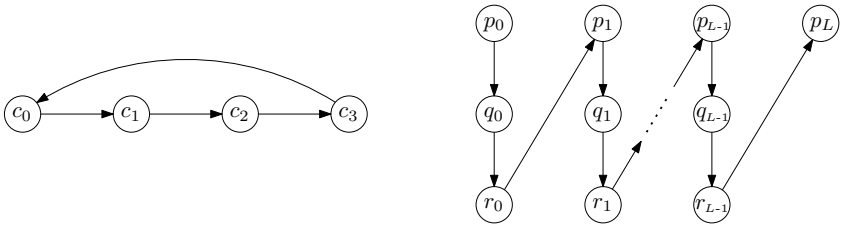

Figure 8: State diagram for $\Gamma\langle h, f\rangle$.

1. $c_0 \rightarrow_{min} c_1$
2. $c_0\ o \rightarrow_{max} c_1$, for $o \in O$
3. $c_1 \rightarrow_{min} c_2$
4. $c_2\ x^v_\pi \rightarrow_{min} c_3\ (x^v_\pi)_{\uparrow_{repl}}$, for $v \in \{0,1\}$ and $\pi \in \bigcup_{l\in[0,L-1]} P(N,l)$
   - $c_2\ v \rightarrow_{min} c_3\ (v)_{\uparrow_{repl}}$, for $v \in \{0,1\}$
   - $c_2\ x^v_j \rightarrow_{min} c_3\ (x^v_j)_{\uparrow_{repl}}$, for $v \in \{0,1\}$, $j \in [1,N]$
5. $c_3 \rightarrow_{min} c_0$
6. $p_t\ x^v_\pi \rightarrow_{min} q_t\ x^?_\pi\ (x^v_\pi)_{\uparrow_{repl}}$, for $t \in [0, L-1]$, $v \in \{0,1\}$, $\pi \in P(N,t)$, s.t. $h \notin Im(\pi)$, $f \notin Im(\pi)$
   - $p_0\ v \rightarrow_{min} q_0\ ?\ (v)_{\uparrow_{repl}}$, for $v \in \{0,1\}$
   - $p_1\ x^v_j \rightarrow_{min} q_1\ x^?_j\ (x^v_j)_{\uparrow_{repl}}$, for $v \in \{0,1\}$, $j \in [1,N]$, s.t. $j \neq h$, $j \neq f$
7. $p_t\ x^v_\pi \rightarrow_{min} q_t\ (x^v_\pi)_{\uparrow_{repl}}$, for $t \in [0, L-1]$, $v \in \{0,1\}$, $\pi \in P(N,t)$, s.t. $h \notin Im(\pi)$, $f \in Im(\pi)$
   - $p_1\ x^v_j \rightarrow_{min} q_1\ (x^v_j)_{\uparrow_{repl}}$, for $v \in \{0,1\}$, $j \in [1,N]$, s.t. $j = f \neq h$
8. $p_t\ x^v_\pi \rightarrow_{min} q_t\ x^?_\pi$, for $t \in [0, L-1]$, $v \in \{0,1\}$, $\pi \in P(N,t)$, s.t. $h \in Im(\pi)$, $f \notin Im(\pi)$

- $p_1\ x_j^v \rightarrow_{min} q_1\ x_j^?$, for $v \in \{0,1\}$, $j \in [1,N]$, s.t. $j = h \neq f$

9. $p_t\ x_\pi^v \rightarrow_{min} q_t$, for $t \in [0, L-1]$, $v \in \{0,1\}$, $\pi \in P(N,t)$, s.t. $h \in Im(\pi)$, $f \in Im(\pi)$
   - $p_1\ x_j^v \rightarrow_{min} q_1$, for $v \in \{0,1\}$, $j \in [1,N]$, s.t. $j = h = f$
10. $q_t \rightarrow_{min} r_t$, for $t \in [0, L-1]$
11. $r_t\ x_\pi^?\ x_\pi^0 \rightarrow_{min} p_{t+1}\ (x^0_{\pi\oplus(f)})_{\downarrow_{repl}}$, for $t \in [0, L-1]$, $\pi \in P(N,t)$
    - $r_0\ ?\ 0 \rightarrow_{min} p_1\ (x_f^0)_{\downarrow_{repl}}$
    - $r_1\ x_j^?\ x_j^0 \rightarrow_{min} p_2\ (x_{jf}^0)_{\downarrow_{repl}}$, for $j \in [1,N]$
12. $r_t\ x_\pi^?\ x_\pi^1 \rightarrow_{min} p_{t+1}\ (x^1_{\pi\oplus(f)})_{\downarrow_{repl}}$, for $t \in [0, L-1]$, $\pi \in P(N,t)$
    - $r_0\ ?\ 1 \rightarrow_{min} p_1\ (x_f^1)_{\downarrow_{repl}}$
    - $r_1\ x_j^?\ x_j^1 \rightarrow_{min} p_2\ (x_{jf}^1)_{\downarrow_{repl}}$, for $j \in [1,N]$
13. $r_t\ x_\pi^? \rightarrow_{min} p_{t+1}\ (x^W_{\pi\oplus(f)})_{\downarrow_{repl}}$, for $t \in [0, L-1]$, $\pi \in P(N,t)$
    - $r_0\ ? \rightarrow_{min} p_1\ (x_f^W)_{\downarrow_{repl}}$
    - $r_1\ x_j^? \rightarrow_{min} p_2\ (x_{jf}^W)_{\downarrow_{repl}}$, for $j \in [1,N]$
14. $r_t\ y \rightarrow_{max} p_{t+1}$, for $t \in [0, L-1]$ and $y \in V$

As we will discuss in Section 7.1, rulesets (6,7,8,9) can be replaced by the following ruleset (15), provided that rulesets (11, 12, 13) are expanded with the additional constraint $f \notin Im(\pi)$.

15. $p_t\ x_\pi^v \rightarrow_{min} q_t\ x_\pi^?\ (x_\pi^v)_{\uparrow_{repl}}$, for $t \in [0, L-1]$, $v \in \{0,1\}$, $\pi \in P(N,t)$
    - $p_0\ v \rightarrow_{min} q_0\ ?\ (v)_{\uparrow_{repl}}$, for $v \in \{0,1\}$
    - $p_1\ x_j^v \rightarrow_{min} q_1\ x_j^?\ (x_j^v)_{\uparrow_{repl}}$, for $v \in \{0,1\}$, $j \in [1,N]$

6.2 Rule sequence R_Ψ

These are the rules for cell ψ_h, the only cell of module Ψ. The state transitions for these sequences are illustrated in Figure 9. State s_{L+1} is the final state for cell ψ_h, i.e. no further transition is possible from this state. For ruleset (1), a message sent up and down by *repl* is an internal message sent to cells γ_{hf} and θ_λ. For ruleset (2), a message sent down by *repl* is an internal message sent to cell θ_λ. These rules are further discussed in Section 7.2.

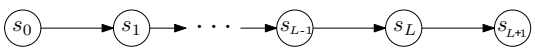

Figure 9: State diagram for the P module Ψ.

1. $s_t\ x_\pi^v \rightarrow_{min} s_{t+1}\ (x_\pi^v)_{\downarrow_{repl}}$, for $t \in [0, L-1]$, $v \in \{0,1\}$, $\pi \in P(N,t)$
   - $s_0\ v \rightarrow_{min} s_1\ (v)_{\downarrow_{repl}}$, for $v \in \{0,1\}$
   - $s_1\ x_j^v \rightarrow_{min} s_2\ (x_j^v)_{\downarrow_{repl}}$, for $v \in \{0,1\}$, $j \in [1,N]$
2. $s_L\ x_\pi^v \rightarrow_{min} s_{L+1}\ (*\ x_\pi^v)_{\downarrow_{repl}}$, for $v \in \{0,1\}$, $\pi \in P(N,L)$
   - $s_2\ x_{jk}^v \rightarrow_{min} s_3\ (*\ x_{jk}^v)_{\downarrow_{repl}}$, for $v \in \{0,1\}$, $j,k \in [1,N]$, $j \neq k$

6.3 Rule sequence R_Θ

These are the rules for the EIG cells of module Θ and belong to Phase I (states d_0 to d_L) and Phase II (states u_0 to u_6). The state transitions for these sequences are illustrated in Figure 10. State u_6 is the final state for all EIG cells, i.e. no further transition is possible from this state. For rulesets (3, 4), a message sent down to $\pi(1)$ is an internal message sent to the child cell $\theta_{\pi(1)}$. These rules are further discussed in Section 7.3.

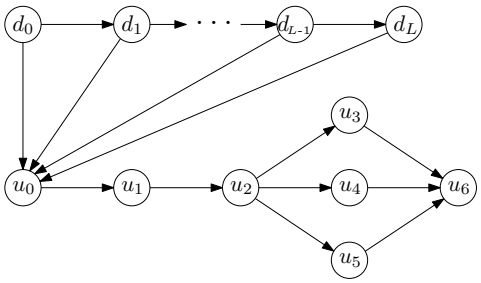

Figure 10: State diagram for the P module Θ.

1. $d_0\ *\ v \rightarrow_{min} u_0\ v'\ v$, for $v \in \{0,1\}$
2. $d_0\ v \rightarrow_{min} d_1\ v'$, for $v \in \{0,1\}$
3. $d_t\ *\ x_\pi^v \rightarrow_{min} u_0\ (*\ x_{\pi(2:t)}^v)_{\downarrow_{\pi(1)}}$, for $t \in [1,L]$, $v \in \{0,1\}$, $\pi \in P(N,t)$
   - $d_1\ *\ x_j^v \rightarrow_{min} u_0\ (*\ v)_{\downarrow_j}$, for $v \in \{0,1\}$, $j \in [1,N]$
   - $d_2\ *\ x_{jk}^v \rightarrow_{min} u_0\ (*\ x_k^v)_{\downarrow_j}$, for $v \in \{0,1\}$, $j,k \in [1,N], j \neq k$
4. $d_t\ x_\pi^v \rightarrow_{min} d_{t+1}\ (x_{\pi(2:t)}^v)_{\downarrow_{\pi(1)}}$, for $t \in [1,L-1]$, $v \in \{0,1\}$, $\pi \in P(N,t)$
   - $d_1\ x_j^v \rightarrow_{min} d_2\ (v)_{\downarrow_j}$, for $v \in \{0,1\}$, $j \in [1,N]$
5. $u_0\ v \rightarrow_{min} u_1\ v$, for $v \in \{0,1\}$
6. $u_1\ 0\ 1 \rightarrow_{max} u_2$
7. $u_1 \rightarrow_{min} u_2$
8. $u_2\ 0 \rightarrow_{max} u_3$
9. $u_2\ 1 \rightarrow_{max} u_4$
10. $u_2 \rightarrow_{min} u_5$
11. $u_3 \rightarrow_{min} u_6\ 0''\ (0)_{\uparrow_{repl}}$
12. $u_4 \rightarrow_{min} u_6\ 1''\ (1)_{\uparrow_{repl}}$
13. $u_5 \rightarrow_{min} u_6\ W''\ (W)_{\uparrow_{repl}}$

7 Program correctness and runtime complexity

Due to space constraints, we only present semi-formal and informal arguments, which at times appeal to the intuition. However, our discussion can be further elaborated into a more formal set of results and proofs.

7.1 Rule sequences $R_\gamma\langle h, f\rangle$ and R'_γ

Consider $t \in [0, L-1]$, a messaging round of Phase I, in the algorithm described in Section 4.1.

Let $\Delta(t)$ be the set of *correctly formatted* objects that can be sent between participants at round t, $\Delta(t) = \{x^v_\pi \mid v \in \{0,1\}, \pi \in P(N,t)\}$. Let $\Delta_i(t)$ be the subset of messages which can be sent by the correct participant $i \in [1,N]$, $\Delta_i(t) = \{x^v_\pi \in \Delta(t) \mid i \notin Im(\pi)\}$ (as noted in Section 4.1, objects which already include the sender's identity will create useless *loops* and should not be included).

Let $\Delta^?(t) = \{x^?_\pi \mid \pi \in P(N,t)\}$ and $\Delta^?_j(t) = \{x^?_\pi \in \Delta^?(t) \mid j \notin Im(\pi)\}$, for $j \in [1,N]$. Set $\Delta^?(t)$ defines *templates*, which describe the format of all correct inter-participant messages possible at round t. Set $\Delta^?_j(t)$ restricts these templates, to the format of correct messages expected from external participant j. An object x^v_π *matches* a template $x^?_{\pi'}$ if they share the underlying permutation, $\pi = \pi'$ (ignoring the actual value $v \in \{0,1\}$).

Consider a module $\Gamma\langle h, f\rangle$, where h is the internal trusted "home" participant and f is the external unreliable "friend-or-foe" participant. This module has two cells, the rule sequence $R_\gamma\langle h, f\rangle$ applies to cell γ_{hf}, and the rule sequence R'_γ to cell γ'_{hf}.

Cell γ'_{hf} is the front-end prepared to receive any kind of messages, from the unreliable cell γ_{fh} (i.e. from external participant f). Rulesets (1,2,3,4,5) collectively ensure the required inter-participant synchronization. Additionally, ruleset (2) deletes all unnecessary objects, and rulesets (4) forwards to γ_{hf} exactly one copy of each correctly formatted message received (that also appears in $\Delta(t)$). Thus, rulesets (1,2,3,4,5) form a filter which ensures that cell γ_{hf} receives external messages at specific steps only and is not "polluted" with duplicate objects or template objects from an unreliable source.

Cell γ_{hf} can only receive trusted messages from the main cell ψ_h and filtered messages from γ'_{hf}; cell γ_{hf} cannot receive any message directly from external participant f, because its external connection is an out-going *simplex* channel.

Consider a set of trusted messages, $\Sigma_h(t)$, sent by the main cell ψ_h to cell γ_{hf}. Assume that $\Sigma_h(t)$ completely describes the EIG level t of participant h, i.e. $\{\pi \mid \exists v \in \{0,1\}, s.t., x^v_\pi \in \Sigma_h(t)\} = P(N,t)$. Rulesets (6,7,8,9) forward $\Sigma_h(t) \cap \Delta_h(t)$ to the external cell γ'_{fh} (i.e. to participant f) and create the templates set $\Delta^?_f(t)$ (for the correct messages expected to be received in turn from participant f), assuming, as above, that $P(N,t)$ equals the permutations set underlying $\Sigma_h(t)$.

Ruleset (6) processes trusted messages in $\Sigma_h(t) \cap \Delta_h(t) \cap \Delta_f(t)$. These messages are sent to f, and they are also used to build the templates set $\Delta^?_f(t)$. Ruleset (7) processes trusted messages in $\Sigma_h(t) \cap \Delta_h(t) \cap \overline{\Delta_f(t)}$. These messages are sent to f, but no templates are built. Ruleset (8) processes trusted messages in $\Sigma_h(t) \cap \overline{\Delta_h(t)} \cap \Delta_f(t)$. These messages are not sent, however, they are used to build the template set templates set $\Delta^?_f(t)$. Ruleset

(9) processes trusted messages in $\Sigma_h(t) \cap \overline{\Delta_h(t)} \cap \overline{\Delta_f(t)}$. These messages are discarded, they are not sent from participant h and no templates are built.

Ruleset (10) defines a one step delay, required for synchronization.

Rulesets (11, 12, 13) attempt to match previously created templates $\Delta^?_f(t)$ against untrusted but correctly formatted objects received from γ'_{fh}. Ruleset (11) matches templates against objects which carry the 0 value and ruleset (12) matches templates against objects which carry the 1 value. If both these matches fail, ruleset (13) assumes a match against the default value W. Finally, the matched object x^v_π is sent to the main cell ψ_h, after appending f to the permutation π, to indicate the sender in a trusted way. This is a critical step, which justifies the complex construction of the module $\Gamma\langle h, f\rangle$, on top of the main cell ψ_h. The usual definitions for channels used in P systems do not offer any protection against impersonation, specifically against participants which claim false identities.

Ruleset (14) performs an additional cell cleanup.

As mentioned in Section 6.1, rulesets (6,7,8,9) can be replaced by the ruleset (15), provided that rulesets (11, 12, 13) are expanded with the additional constraint $f \notin Im(\pi)$. This simplified ruleset still functions correctly, but slightly increases the messaging complexity of the original algorithm. According to the new ruleset (15), cell γ_{hf} sends out all objects in $\Sigma_h(t)$ (not only those in $\Sigma_h(t) \cap \Delta_h(t)$) and creates the larger hence weaker templates set $\Delta^?(t)$. However, the new versions of rulesets (11,12,13) still filter correctly, because the combination of the template set $\Delta^?(t)$ with the additional constraint $f \notin Im(\pi)$ logically recreates the stronger template set $\Delta^?_f(t)$.

7.2 Rule sequence R_Ψ

Ruleset (1) takes one object x^v_π, where π is a permutation of N of length up to $L-1$, and sends one copy of this object to each cell γ_{hf}. Cell γ_{hf} will forward this copy, via its external connection, to another participant, as discussed in Section 7.1. Simultaneously, ruleset (1) sends down one copy of the same object x^v_π, to the EIG root cell θ_λ. Cell θ_λ uses the objects, that are *not* accompanied by asterisks, to populate levels 0 to $L-1$ of the EIG tree with top-down values, as discussed in Section 7.3.

Ruleset (2) takes one object x^v_π, where π is a permutation of N of length L and sends down one copy of this object, accompanied by one asterisk (*), to the EIG root cell θ_λ. Cell θ_λ uses the objects, that are accompanied by asterisks, to populate the last level of the EIG tree with top-down values, as discussed in Section 7.3.

These two rulesets, (1) and (2), run as part of a send-receive cycle, which is repeated L times. Initially, cell ψ_h starts with one object $v = x^v_{()}$, representing its initial decision value. This object is processed by ruleset (1), as described above, i.e. is sent to all other participants and used as the first top-down value in the EIG tree. At the end of the first messaging round, cell ψ_h receives a set of messages of the form x^u_f, where f is the index of another participant and u is f's initial decision value. These messages are simultaneously processed by ruleset (1), which continues the cycle.

For the first $L-1$ times, the received messages are processed by ruleset (1), which continues the send-receive cycle. The messages received at the end of the last cycle, L, are processed by ruleset (2), which stops the send-receive cycle.

The asterisks sent down by ruleset (2) accompanies the top-down values of the last EIG level. As discussed in Section 7.3, this triggers the bottom-up evaluation of the EIG tree. The bottom-up value evaluated by the root EIG cell θ_λ represents the final decision of this participant and a copy of it is sent to the main cell ψ_h.

7.3 Rule sequence R_Θ

Rules starting from states d_0 to d_L belong to Phase I and ensure that the EIG nodes are properly filled with top-down values. The asterisk object (*) accompanies the last round values, until these reach the leaves, ensuring a proper transition to the next phase. Rules starting from states u_0 to u_6 belong to Phase II and evaluate bottom-up values. Figure 10 shows a bird's eye view of the state transitions of this rule sequence.

An inductive argument can used to describe the successive filling of the first L EIG levels with top-down values. Consider $t \in [0, L-1]$ and assume the following induction hypothesis: (a) all EIG cells at level $k \in [0, t]$ are in state d_{t-k}; and (b) all EIG cells below level t are in state d_0. Obviously, this hypothesis holds initially, for $t = 0$.

Consider now that (c) the root EIG cell receives the object x^v_π, where $v \in \{0, 1\}$, $\pi \in P(N, t)$, without any accompanying asterisk object. Rulesets (2, 4) ensure that the value v is carried over by successively trimmed objects $x^v_\pi = x^v_{\pi_1\pi_2\ldots\pi_t}, x^v_{\pi_2\ldots\pi_t}, \ldots, x^v_{()} = v$, and routed downwards, along a path $\lambda, \pi_1, \ldots, \pi_{t-1}, \pi_t$. All cells involved in this transfer, $\lambda, \pi_1, \ldots, \pi_{t-1}, \pi_t$, successively change their states to d_{t+1}, d_t, ..., d_2, d_1, respectively. Additionally, by ruleset (2), the last cell in the path keeps its original top-down value v in the alternate form v'.

A slightly different argument is needed to describe the filling of the last EIG level, L. Consider a similar context as above, for $t = L$, where condition (a) still holds and condition (b) is true, but irrelevant, because there are no cells below level L.

Consider a modified condition (c^*), where the root cell additionally receives one asterisk for each object x^v_π. Rulesets (3, 1) ensure that this asterisk accompanies the value v, along a similar path, until they reach the leaf level L. All cells involved in this transfer successively change their state to u_0. Additionally, by ruleset (1), the last cell in the path (the leaf) keeps both its original top-down value v and its alternate form v'. Similar transitions eventuate synchronously for all leaf cells, assuming that the root cell is timely filled with the objects required to completely fill the EIG tree, successively, level by level.

State u_0 is the initial state of Phase II, the bottom-up evaluation. This process starts from the EIG leaf cells and continues upwards, level by level. Each cell applies a local voting scheme, to the values received from its child cells, if it is not a leaf, or, if it is a leaf, to its single value v. Ruleset (5) ensures that the bottom-up evaluation does not start prematurely. Rule (6) cancels matching 0 and 1 pairs. If any 0's remain, the cell decides on 0, by way of rules (8, 11). If any 1's remain, the cell decides on 1, by way of rules (9, 12). Otherwise, the cell decides on the default hardcoded value W, by way of rules (10, 13). Any decision $v \in \{0, 1\}$ is sent up to the parent cell and also recorded locally in the alternate form v''. Using an inductive argument, the root cell takes the expected decision and sends it up to the main cell.

7.4 Runtime complexity

Revisiting the above arguments and counting the steps, we obtain the following result, which highlights the runtime complexity of our system.

Theorem 5. *This EIG tree based Byzantine algorithm takes* $9L+6$ *steps, where Phase I takes* $5L+2$ *steps and Phase II takes* $4L+4$ *steps.*

Proof. In Phase I, each messaging round between two main cells ψ_h and ψ_f takes 4 steps, along the following route: ψ_h, γ_{hf}, γ'_{fh}, γ_{fh}, ψ_f. Therefore, L messaging rounds take $4L$ steps, which also covers the time required to fill the first $L-1$ levels of the EIG tree. After the last round messages are received, $L+2$ steps are further required to fill level L of each EIG tree. Thus, Phase I takes $5L+2$ steps in total.

The bottom-up evaluation takes 4 steps for each level and the EIG tree has $L+1$ levels. Thus, Phase II takes $4(L+1)$ steps to complete its evaluation. □

7.5 Sample run

Figures 11, 12 and 13 offer additional insight on the overall behavior of module Ω, via partial traces of our main sample scenario, described in detail in Section 4 and illustrated in Figures 2, 3 and 4.

Figure 11 illustrates traces, describing the messaging interaction between participants 2 and 3. The following cells are included: θ_{λ_2}, ψ_2, γ_{23}, γ'_{23} (for participant 2); and γ'_{32}, γ_{32}, ψ_3, θ_{λ_3} (for participant 3). After ten steps, all port cells reach their final states.

Figure 12 and Figure 13 illustrate the top-down and bottom-up evaluations, respectively, of the EIG tree of participant 2. The following cells are included: $\theta_{1.2}$, $\theta_{1.3}$, $\theta_{1.4}$, θ_1, θ_λ, θ_2, θ_3, θ_4. All EIG cells end in the final state u_6. In the last step, the root EIG cell, θ_λ, decides on 0 and, simultaneously, records this as 0" and sends one 0 up, to the main cell ψ_2 (cell ψ_2 is not illustrated in these figures).

This sample run ends in 24 steps, which is consistent with the runtime complexity given by Theorem 5, i.e. $9L+6$ steps, where $L=2$.

8 Conclusion

In this paper, we have proposed a new modular framework for designing P system programs and used it to investigate the Byzantine agreement problem. Our modular framework allows encapsulations, information hiding and modular composition. We believe that our solution of the Byzantine agreement problem is the first complete P system solution for this problem (which effectively lists all its rules).

Our P program was also successfully tested on our P system simulator, for a fair number of scenarios, including various combinations of Byzantine behaviors, such as wrong messages, incorrectly formatted messages, extra messages, missing messages and out-of-sync messages.

Our investigation leaves open a number of interesting and challenging problems. The Byzantine agreement algorithm is an interactive algorithm which solves its problem only

in synchronous networks, where all interactions must eventuate within fixed time limits. Will an algorithm built via an universalization technique meet such requirements? Can we achieve a Byzantine agreement using only duplex channels (without any simplex channels)? The number of cells and rules of our P program for the Byzantine agreement grows exponentially in N and L and the message size is larger than optimal. Can we reduce the space complexity of our messaging phase? In our program, all the cells must be created and connected before our algorithm starts. Is it possible to solve the same problem with a fixed number of cells? Otherwise, is it possible to solve the same problem starting with a fixed number of cells, and develop a dynamically growing EIG tree? Is it possible to solve the same problem with a fixed number of rules? Can we design P system programs for other Byzantine agreement algorithms, not EIG-based, for example using reliable broadcasts? Can we extend our P system programs to cover $2F + 1$ connected graphs, but not necessarily complete?

Acknowledgments

The authors wish to thank John Morris and the four anonymous reviewers for detailed comments and feedback that helped us improve the paper.

References

[1] Michael Abd-El-Malek, Gregory R. Ganger, Garth R. Goodson, Michael K. Reiter, and Jay J. Wylie. Fault-scalable Byzantine fault-tolerant services. In Andrew Herbert and Kenneth P. Birman, editors, *SOSP*, pages 59–74. ACM, 2005.

[2] Artiom Alhazov, Maurice Margenstern, and Sergey Verlan. Fast synchronization in P systems. In David W. Corne, Pierluigi Frisco, Gheorghe Păun, Grzegorz Rozenberg, and Arto Salomaa, editors, *Workshop on Membrane Computing*, volume 5391 of *Lecture Notes in Computer Science*, pages 118–128. Springer, 2008.

[3] Michael Ben-Or and Avinatan Hassidim. Fast quantum Byzantine agreement. In Harold N. Gabow and Ronald Fagin, editors, *STOC*, pages 481–485. ACM, 2005.

[4] Francesco Bernardini, Marian Gheorghe, Maurice Margenstern, and Sergey Verlan. How to synchronize the activity of all components of a P system? *Int. J. Found. Comput. Sci.*, 19(5):1183–1198, 2008.

[5] Christian Cachin, Klaus Kursawe, and Victor Shoup. Random oracles in constantinople: Practical asynchronous Byzantine agreement using cryptography. *J. Cryptology*, 18(3):219–246, 2005.

[6] Miguel Castro and Barbara Liskov. Practical Byzantine fault tolerance and proactive recovery. *ACM Trans. Comput. Syst.*, 20(4):398–461, 2002.

[7] Gabriel Ciobanu. Distributed algorithms over communicating membrane systems. *Biosystems*, 70(2):123–133, 2003.

[8] Gabriel Ciobanu, Rahul Desai, and Akash Kumar. Membrane systems and distributed computing. In Gheorghe Păun, Grzegorz Rozenberg, Arto Salomaa, and Claudio Zandron, editors, *WMC-CdeA*, volume 2597 of *Lecture Notes in Computer Science*, pages 187–202. Springer, 2002.

[9] Michael J. Dinneen, Yun-Bum Kim, and Radu Nicolescu. New solutions to the firing squad synchronization problems for neural and hyperdag P systems. *EPTCS*, 11:107–122, 2009.

[10] Leslie Lamport, Robert E. Shostak, and Marshall C. Pease. The Byzantine generals problem. *ACM Trans. Program. Lang. Syst.*, 4(3):382–401, 1982.

[11] Nancy A. Lynch. *Distributed Algorithms*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 1996.

[12] Jean-Philippe Martin and Lorenzo Alvisi. Fast Byzantine consensus. *IEEE Trans. Dependable Sec. Comput.*, 3(3):202–215, 2006.

[13] Radu Nicolescu, Michael J. Dinneen, and Yun-Bum Kim. Structured modelling with hyperdag P systems: Part A. Report CDMTCS-342, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, Auckland, New Zealand, December 2008. `http://www.cs.auckland.ac.nz/CDMTCS/researchreports/342hyperdagA.pdf`

[14] Radu Nicolescu, Michael J. Dinneen, and Yun-Bum Kim. Discovering the membrane topology of hyperdag P systems. In Gheorghe Păun, Mario J. Pérez-Jiménez, Agustín Riscos-Núñez, Grzegorz Rozenberg, and Arto Salomaa, editors, *Membrane Computing, Tenth International Workshop, WMC 2009*, volume 5957 of *Lecture Notes in Computer Science*, pages 410–435. Springer, 2009.

[15] Radu Nicolescu, Michael J. Dinneen, and Yun-Bum Kim. Structured modelling with hyperdag P systems: Part A. In Rosa Gutiérrez-Escudero, Miguel A. Gutiérrez-Naranjo, Gheorghe Păun, and Ignacio Pérez-Hurtado, editors, *Membrane Computing, Seventh Brainstorming Week, BWMC 2009, Sevilla, Spain, February 2-6, 2009*, volume 2, pages 85–107. Universidad de Sevilla, 2009.

[16] Radu Nicolescu, Michael J. Dinneen, and Yun-Bum Kim. Structured modelling with hyperdag P systems: Part B. Report CDMTCS-373, Centre for Discrete Mathematics and Theoretical Computer Science, The University of Auckland, Auckland, New Zealand, October 2009. `http://www.cs.auckland.ac.nz/CDMTCS/researchreports/373hP_B.pdf`

[17] Gheorghe Păun. *Membrane Computing: An Introduction*. Springer-Verlag New York, Inc., Secaucus, NJ, USA, 2002.

[18] Gheorghe Păun. Introduction to membrane computing. In Gabriel Ciobanu, Mario J. Pérez-Jiménez, and Gheorghe Păun, editors, *Applications of Membrane Computing*, Natural Computing Series, pages 1–42. Springer, 2006.

[19] Gheorghe Păun and Mario J. Pérez-Jiménez. Solving problems in a distributed way in membrane computing: dP systems. *International Journal of Computers, Communications and Control*, 2 (2010, to appear), 2010.

[20] Marshall C. Pease, Robert E. Shostak, and Leslie Lamport. Reaching agreement in the presence of faults. *J. ACM*, 27(2):228–234, 1980.

[21] Wikipedia.org Byzantine fault tolerance—Wikipedia, the free encyclopedia, 2009. `http://en.wikipedia.org/w/index.php?title=Byzantine_fault_tolerance&oldid=330075795`, [Online; accessed 9-January-2010].

[22] Wikipedia.org Quantum byzantine agreement—Wikipedia, the free encyclopedia, 2009. `http://en.wikipedia.org/w/index.php?title=Quantum_Byzantine_agreement&oldid=335090911`, [Online; accessed 9-January-2010].

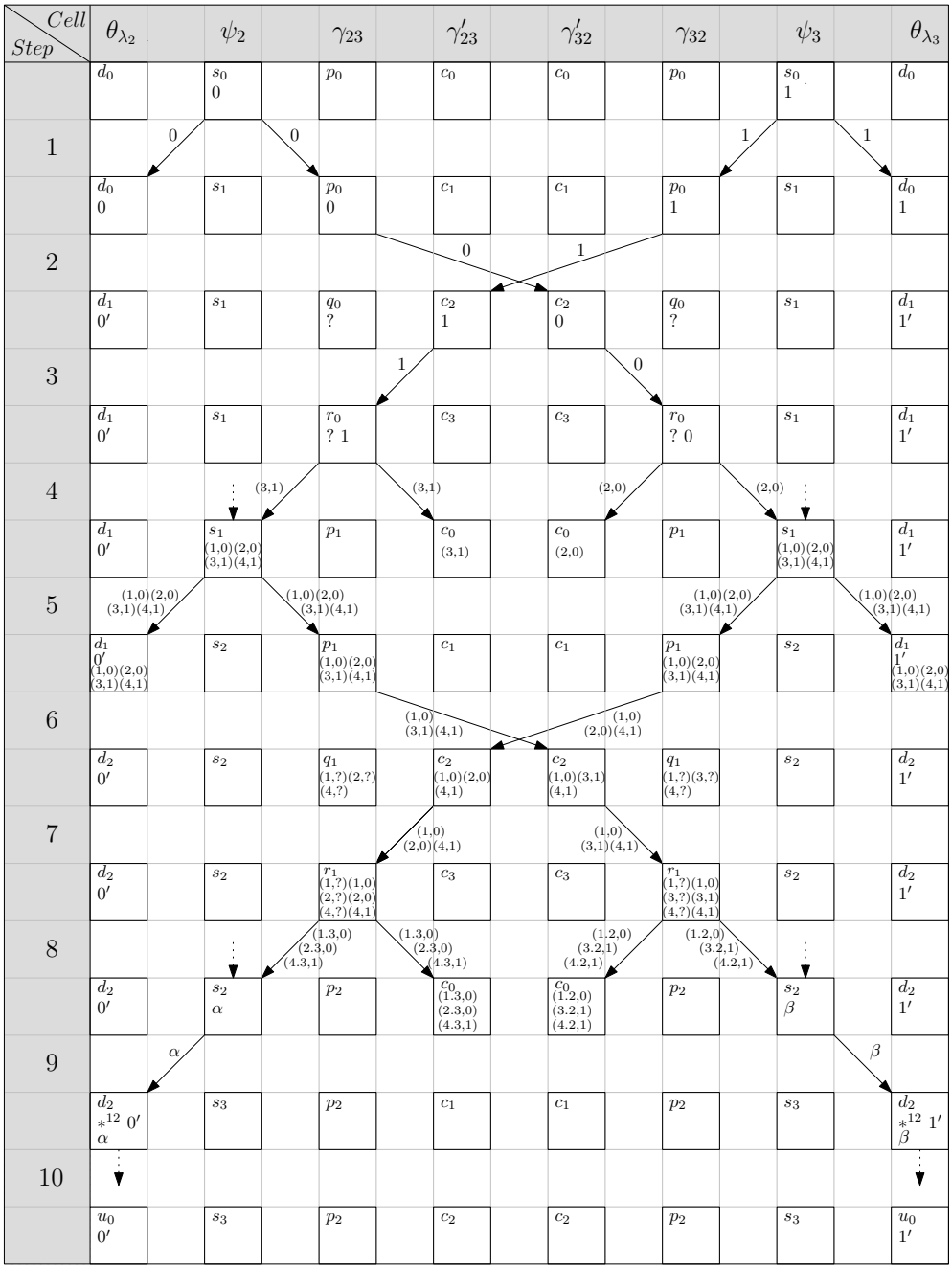

Figure 11: Traces of the message phase between participants 2 and 3 (fragments). Here, α = { (1.2,0), (1.3,0), (1.4,1), (2.1,0), (2.3,0), (2.4,0), (3.1,1), (3.2,1), (3.4,1), (4.1,1), (4.2,1), (4.3,1) } and β = { (1.2,0), (1.3,0), (1.4,1), (2.1,0), (2.3,0), (2.4,0), (3.1,0), (3.2,1), (3.4,1), (4.1,1), (4.2,1), (4.3,1) }.

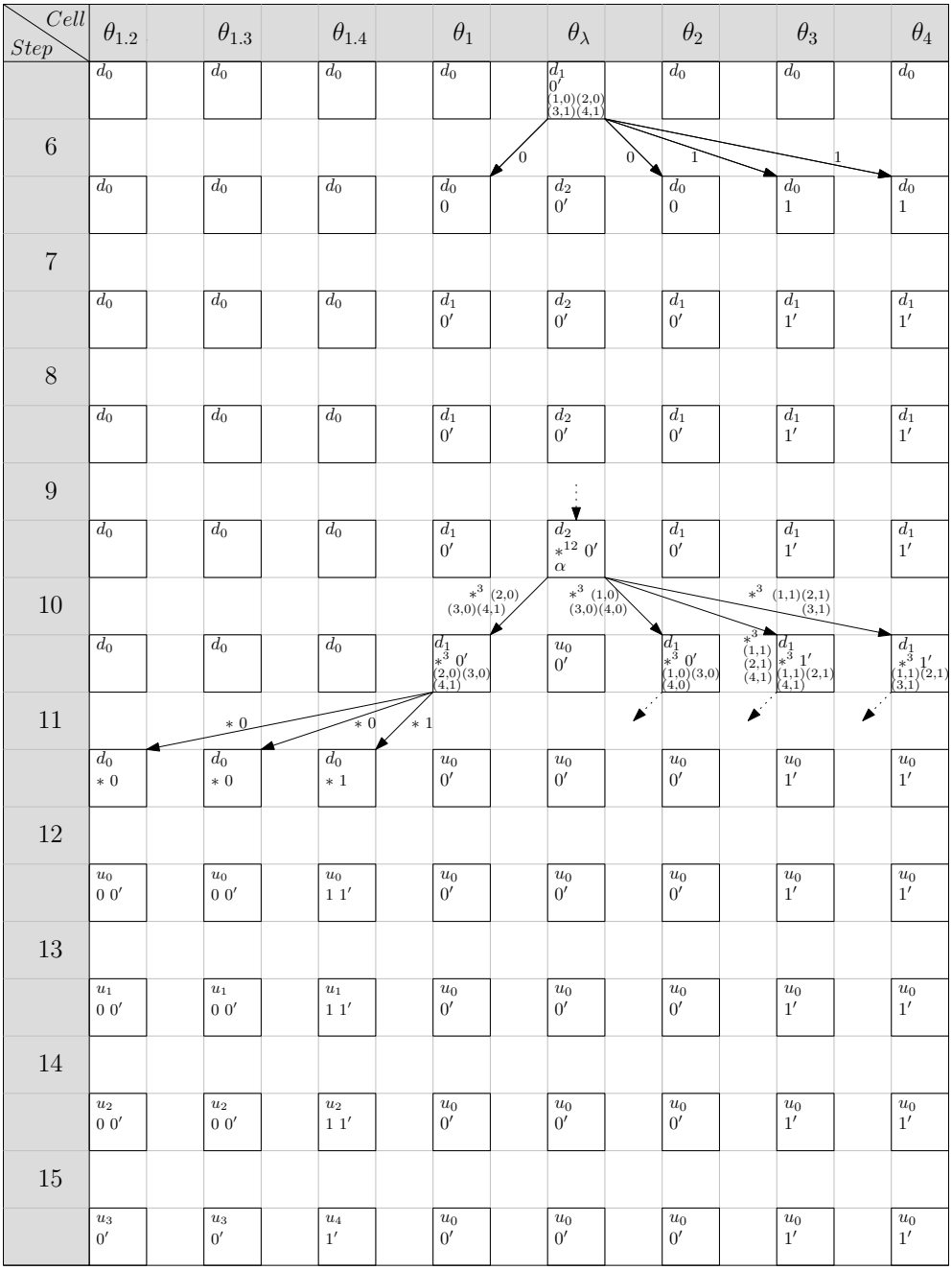

Figure 12: Traces of the top-down evaluation of the EIG tree of participant 2 (fragments). Here, α = { (1.2,0), (1.3,0), (1.4,1), (2.1,0), (2.3,0), (2.4,0), (3.1,1), (3.2,1), (3.4,1), (4.1,1), (4.2,1), (4.3,1) }.

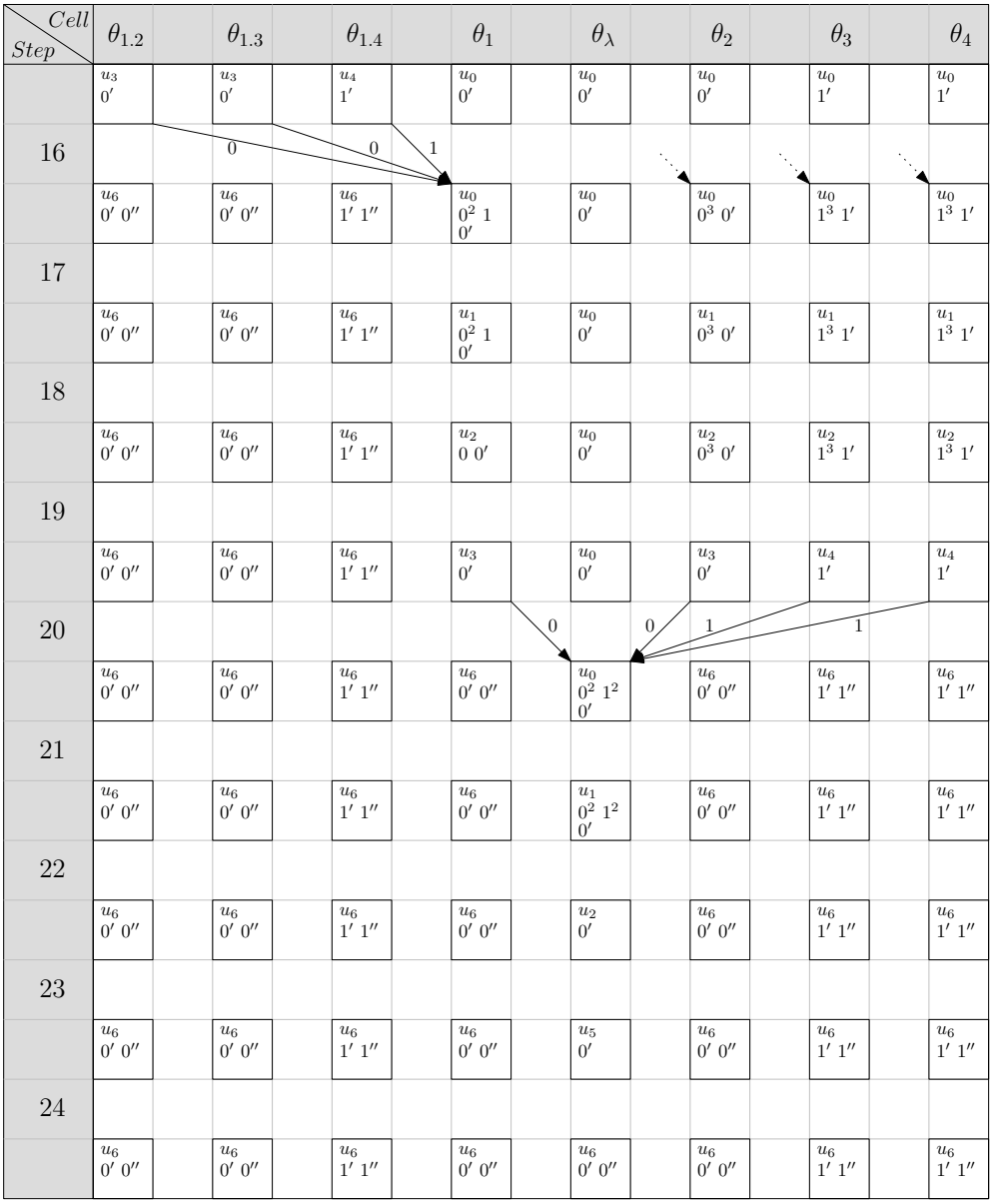

Figure 13: Traces of the bottom-up evaluation of the EIG tree of participant 2 (fragments).