

The Wake Up and Report Problem is Time-Equivalent to the Firing Squad Synchronization Problem

Darin Goldstein* and Nick Meyer†

Abstract

We consider several problems relating to strongly-connected *directed* networks of identical finite-state processors that work synchronously in discrete time steps. The conceptually simplest of these problems is the Wake Up and Report Problem; this is the problem of having a unique “root” processor send a signal to all other processors in the network and then enter a special “done” state only when all other processors have received the signal. The most difficult of the problems we consider is the classic Firing Squad Synchronization Problem; this is the much-studied problem of achieving macro-synchronization in a network given micro-synchronization. We show via a complex algorithmic application of the “snake” data structure first introduced in Even, Litman, and Winkler[6] that these two problems in particular are asymptotically time-equivalent up to a constant factor. This result leads immediately to the inclusion of several other related problems into this new asymptotic time-class.

1 Introduction

In this paper, we study a number of fundamental protocols that run on *directed* networks of identical synchronous finite-state automata. Two of the most important are the Wake Up and Report Problem (or WURP, also known equivalently in the literature as the “Broadcast With Echo” Problem¹), conceptually the simplest problem to solve, and the Firing Squad Synchronization Problem (or FSSP), probably the most difficult.

Generally speaking, the bulk of researchers studying theoretical protocols on such networks tend to focus on the seemingly more difficult and intricate FSSP which has a forty-year history of research behind it. The WURP, though obviously important, seems to receive

less attention because of its apparent simplicity. The recent research trend (e.g. [6], [13]) seems to be for a group to discover a clever solution to the FSSP and then notice that the solution also solves a number of other problems including the WURP. Our main result is the introduction of a new time-class of asymptotically equivalent network problems which includes both the WURP and the FSSP. This result makes it possible for researchers to reverse the trend. Instead of working on the “difficult” problems and noting that the solutions solve the “easy” ones, researchers can now focus their efforts on “easy” problems and note via the result in this paper that asymptotically those solutions also solve the “difficult” ones. One can, of course, look at this benefit from the opposite direction: a *lower* asymptotic time bound for one of the “difficult” problems also applies to the “easy” ones.

1.1 The Network Model. We consider strongly-connected *directed* networks of identical synchronous finite-state automata with in- and out-degree bounded by a constant. These automata are meant to model very small, very fast processors. The network itself has unknown topology and potentially unbounded size N . Throughout this paper, we use the term “-port” to refer to one of a number of unidirectional conduits through which constant-size messages may pass from one processor to another. An *in-port* to a processor will allow messages to flow unidirectionally in towards the processor. *Out-port* is defined similarly. We assume throughout that the number of in-ports and out-ports for each processor is uniformly bounded by a network constant δ . The network is formed by connecting out-ports from automata to the in-ports of other automata with wires. Not all in-ports or out-ports of a given automaton need necessarily be connected to other automata. Note that even though the communication links are unidirectional, a pair of processors is allowed to be connected with two communication links, one in either direction, simulating a bidirectional link.

We assume that each processor in the network is initially in a special “quiescent” state, in which, at each time-step, the processor sends a “blank” character

*Computer Science Department, Cal State Long Beach, darin@cecs.csulb.edu

†Mathematics Department, U.C. Berkeley, meyer@math.berkeley.edu

¹The authors had a tough choice to make between WURP and BWEP. We chose WURP. The reader is welcome to weigh in on his acronym of choice.

through all of its out-ports. A processor remains in the quiescent state until a non-blank character is received by one of its in-ports.

The network has a global clock, the pulses between which each processor performs its computations. Processors synchronously, within a single global clock pulse, perform the following actions in order: read in the inputs from each of their in-ports, process their individual state changes, and prepare and broadcast their outputs.

Our network structure is specifically designed to model the practical situation of many small and fast processors with only the capacity for reliable one-way communication. Aside from being intrinsically mathematically interesting in its own right, this particular model applies to many practical situations as well. Unidirectional communication commonly occurs with one-way radio networks (e.g. the GPS satellites currently in orbit about Earth, encrypted military networks, etc.), VLSI design, and bidirectional networks in which in-ports or out-ports are allowed to undergo shutdown failures.

The reason for modeling the processors by identical finite-state automata is simple. In practice, many network protocols are expected to run extremely fast. One particular reason for this is that the network topology or size might change if the protocol takes too long, thereby potentially rendering the computation obsolete or incorrect. (For example, if the network is attempting to synchronize itself and a processor is randomly added to the topology of the network in the middle of the computation, it is likely to throw off the timing of the synchronization.) It is therefore assumed that the processors involved will not have time to access a large memory cache. Commonly, a memory access can take orders of magnitude longer than a simple state-change processor calculation. The current technological trend is to merge the memory functions that one generally associates with a computing machine into the processor itself (for further discussion, see [3], [9].) Hence, we model the processors by identical finite-state automata.

The protocols described below are presumed to begin when a certain processor is signaled by some outside source. We call this processor the *root*, and assume that every processor knows whether or not it is the root. (It is possible to do this by specifying a “root subset” of the state space of the automata that the root is uniquely allowed to use. The root’s state must always lie in this special root subset, and no non-root processor’s state can ever be in the root subset.) A protocol ends when the root enters a special terminal state indicating that the computation has successfully completed. In our computational model, we calculate

the time-complexity of a protocol in terms of the total number of global time steps between initiation and termination. Of course, the aim is to minimize this time-complexity.

Finally, for ease of exposition, we define several statistics associated with a given network:

DEFINITION 1.1. *Let distance in the network be defined in the obvious way; i.e., define $d(A, A')$, the distance from processor A to processor A' , to be the length (in number of edges) of a minimal-length directed path from processor A to processor A' . (Note that we are on a directed network, so $d(A, A')$ might not equal $d(A', A)$.)*

We define $\Gamma := \max_{\{\text{processors } A\}} d(\text{root}, A)$, $\Gamma' := \max_{\{\text{processors } A\}} d(A, \text{root})$, and $D := \max_{\{\text{all processors } A, B\}} d(A, B)$.

Note that if we start a breadth-first search at the root, then Γ is the length of the longest branch in the resulting BFS tree.

We point out that none of the finite-state processors comprising our network may be able to store Γ, Γ' , or D , as they may be arbitrarily large.

2 Description and Brief History of Prior Research

Research on networks of identical finite-state automata has been going on for decades, and the body of literature on the subject is enormous. Aside from the two problems presented below, the literature deals with issues such as leader election, mutual exclusion, and network traversal. See [1], [2], [5], and [15] for a *small* sampling. The remainder of this section will be devoted to a description and brief history of the prior work relating to the two main problems we address in this paper, the Wake Up and Report Problem and the Firing Squad Synchronization Problem.

2.1 The Wake Up and Report Problem.

Conceptually, the Wake Up and Report Problem (or WURP) is the simpler of the two problems we will consider in this section. We begin with a network of quiescent processors. Some outside agent nudges the root processor to commence the Wake Up and Report Protocol. The root may terminate the protocol after every processor in the network has been nudged out of the quiescent state. The objective of the protocol is therefore to inform the root in the fastest possible time that every processor has “awakened” from its quiescent state. This is equivalent to the Broadcast and Echo demand that the root be assured that every processor in the network has received a unique signal sent out upon protocol initiation.

As mentioned previously, this problem has not been studied to the same extent as the Firing Squad Synchronization Problem below. The best solution currently known is the $O(ND)$ protocol outlined in Ostrovsky and Wilkerson's 1995 paper[13]. Aside from the references to the Wake Up and Report problem in Even, Litman, and Winkler[6] and Ostrovsky and Wilkerson[13] (both papers mainly dealt with the more difficult FSSP), references to the Wake Up and Report Problem can be found with variations in the network and processor assumptions under the guise of the Broadcast and Echo Problem in Afek and Gafni[1], Propagation Information with Feedback in Segall[14], and Echo protocols in Chang[4]. We should take care to mention that in some of these references, the presented protocols were more concerned with message complexity than time complexity. This paper will concentrate exclusively on the time complexity aspect of the problem.

The only fact about the WURP that we will need is the following intuitively obvious lemma, which states that in order for a WURP protocol to terminate correctly on every network, the root must have had time to both send out and receive a message from every other processor in the network.

LEMMA 2.1. *The running time of any WURP protocol must be greater than $\max(\Gamma, \Gamma')$.*

Proof². Fix some Wake Up and Report Protocol, fix a network, and let t be the running time of the protocol on this particular network. Assume $t \leq \Gamma$. Fix a processor G such that $d(\text{root}, G) = \Gamma$. Now, construct a new network from the old, replacing G with a length-2 chain of processors (call them G_1 and G_2), such that the inputs of G enter the chain at G_1 and the outputs of G exit the chain at G_2 .

Now, the computational transcript of the root in our new network is identical to that of our old network, up through time Γ ; hence, our protocol should once again terminate at time t . But this leads to a contradiction – since $d(\text{root}, G_2) = \Gamma + 1$, it is impossible for processor G_2 to be awakened by time t .

Now assume $t \leq \Gamma'$. Fix a processor G' such that $d(G', \text{root}) = \Gamma'$; define t_0 to be $d(\text{root}, G')$. Let D be the diameter of the network. Construct a new network from the old, replacing G' with a chain of $D+1$ processors (call them G'_1 through G'_{D+1}), such that the inputs of G' enter the chain at G'_1 and the outputs of G' exit the chain at G'_{D+1} . (See Figure 1.) Clearly, the computational transcript of the root is identical to that

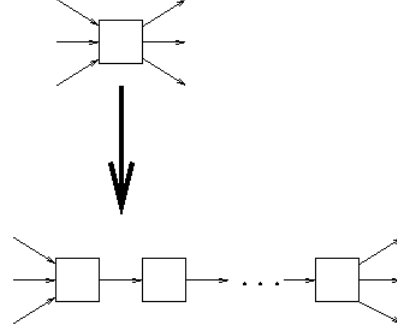


Figure 1: The replacement of a processor with a sequence of processors

with our old network, up through time $t_0 + \Gamma' - 1$, which is certainly greater than or equal to Γ' ; hence, as before, the protocol will terminate at time Γ' . But this leads to a contradiction – since $d(\text{root}, G'_{D+1}) = t_0 + D > \Gamma'$, it is impossible for processor G'_{D+1} to be awakened by time Γ' . \square

This lemma leads to an interesting corollary that, once again, is obviously true on an intuitive level but worth proving explicitly. This Corollary makes explicit the fact that either Γ or Γ' must be at least $\frac{D}{2}$. (Note that it is easily possible to construct families of networks where $\Gamma = O(\log D)$ or $\Gamma' = O(\log D)$. Otherwise, the little substance in this Corollary would disappear entirely.)

COROLLARY 2.1. *Any protocol that solves the Wake Up and Report Problem must take time at least $\frac{D}{2}$, where D is the diameter of the network. Hence, any such protocol has time-complexity $\Omega(D)$.*

Proof. Fix a WURP protocol and a network. Assume the protocol has running time t on the given network; fix processors A and B in our network such that $d(A, B) = D$. Then, from Lemma 2.1, we have $D \leq d(A, \text{root}) + d(\text{root}, B) \leq \Gamma + \Gamma' \leq 2t$, from which the result follows. \square

2.2 The Firing Squad Synchronization Problem. Again, we have a network as described previously. An outside agent nudges the root out of its quiescence. The goal of a Firing Squad Synchronization Protocol is to get every processor in the network to *simultaneously* enter the the same special FIRING state. Termination of the protocol occurs when the root (and hence the rest of the network) has FIRED. This protocol is useful for global calculations which require all processors to begin computation simultaneously. What makes this problem fiendishly difficult is the finite-state property of the au-

²This proof can be skipped without much loss to the reader; we bother to prove this intuitively obvious lemma rigorously because it will be crucial to the proof of our protocol's correctness.

tomata: no single processor has the memory capacity to determine the size or topology of a large enough portion of the network. In general, processors do not even necessarily have the ability to assign themselves unique names if the network is large enough (as it may well be.)

The Firing Squad Synchronization Problem (or FSSP) has a rich history; solutions to various subproblems have been discovered over a period of decades. J. Mazoyer provides an overview of the problem (up to 1986, at least) in addition to some of its history in [10]. We summarize the history here. J. Myhill introduced the problem in 1957, though the first published reference is [12] from 1962. J. McCarthy and M. Minsky first solved the problem for the bidirectional line in [11]. Later, the problem was considered for directed networks of automata. Honda and Nishitani[7] and Kobayashi[8] solved the FSSP for specific subtypes of networks, the directed ring and the “ring-of-trees” – that is, networks which include a loop, containing the root, whose length is at least as great as the maximum distance from the root to any processor. Even, Litman, and Winkler[6] used this, and their invention of network-traversing “snakes” (see below) to create a protocol that would fire any strongly-connected directed network in $O(N^2)$ time. Ostrovsky and Wilkerson[13] were able to improve this to $O(ND)$ in 1995, which remains the best today.

3 Data Structures

Space does not permit us to expand upon the fascinating history outlined in Section 2.2; in particular, we cannot present the various solutions to the FSSP that have been found over the years. Our protocol does make use of several data structures discovered and refined by those mentioned above; we review those structures here.

We first describe how to implement data structures of differing “speeds”. Then we briefly discuss tokens, various types of snakes, and the loops that snakes generate and mark. Finally, we define the “ring-of-trees” network structure, present a way to impose this structure on the network (given a sufficiently long marked loop), and discuss how this will enable us to fire the network quickly.

3.1 Speed. The protocol about to be presented makes use of several computational constructs (to be described below), each of which is assigned a certain characteristic “speed.”³ This is *not* to say that certain messages move faster through the network than others.

³The concept of speed was introduced by McCarthy and Minsky [11] in a paper in which they presented the first published solution to the FSSP on a bidirectional line of processors.

All computations and outputs are strictly synchronous with respect to the global network clock.

In the protocol that follows, the speeds that we utilize are speed-1, and speed-4. The method by which we implement a speed is as follows: A speed-1 construct will enter a processor. It will then remain there for 4 global clock ticks. At the 4th clock tick, it will proceed along its designated path. A speed-4 construct, on the other hand, will wait only for one tick. Thus, in reality, the implementation specifies that a speed-1 construct moves 4 times slower than a speed-4 construct.

3.2 Tokens. Tokens are the simplest data structure possible on networks of finite-state machine. They should be thought of as markers that can be passed from one processor to another via the edges of the network. The token concept has been in use since the first solution to the bidirectional line [11]. The general token behaviors outlined below have also been utilized in other papers (e.g. [6] and [13]), though our definition of “breadth-first” and “loop” tokens below are original to the best of our knowledge.

We employ two main varieties of tokens. *Breadth-first tokens* can be thought of as moving within a “breadth-first-search tree” in the following sense: We arrange it so that each relevant processor in the network has a “parent” marker associated with one of its in-ports⁴. We then declare that a breadth-first token will only be accepted by a given processor when either (a) the processor creates the token, in which case the processor will not have a parent in-port, or (b) the token comes through the processor’s parent in-port. If a breadth-first token is received through a non-parent in-port or by a quiescent processor, it is ignored. Breadth-first tokens are passed out of every out-port; thus, breadth-first tokens multiply in number as time goes on (as long as they stay within the confines of the breadth-first-search tree.) In summary, if a breadth-first token is created at the root of its associated breadth-first tree, then t time steps later there will be a token at each processor that is a distance of t from the root, and none elsewhere. (If the tree has length less than t , of course, there will be no tokens anywhere.)

Loop tokens travel along a specified marked loop within the network. (How loops get marked is described in Section 3.3.) A processor on the loop that receives a loop token simply passes it to the next processor on the loop. In this paper, the root (which will be on every loop we mark) will create every loop token we use; t time steps after its creation, therefore, a loop token will

⁴The method by which various breadth-first-search trees are constructed by snakes, as well as how each processor designates its “parent” in-port, is discussed in Section 5.

be t processors away from the root, along the marked loop. When any loop token reaches the root again, it is absorbed (i.e., not sent around again).

Note that tokens can only carry along with them a constant (very small) amount of information since they are only of constant size. The next data structure takes care of this problem.

3.3 Snakes. The concept of a data-carrying *snake* was invented by Even, Litman, and Winkler in [6]. Snakes are the solution to the problem of the limited data-carrying capabilities of tokens. A snake is capable of carrying an arbitrarily large amount of data, but for this reason, it must reside in a collection of adjacent processors rather than a single processor.

A “snake” is a string – which may be arbitrarily long – made from an alphabet of $2\delta + 1$ characters, namely δ head characters, δ body characters, and a unique tail character. (Recall that δ is a fixed constant of the network.) The characters comprising the string are stored in adjacent processors, one character per processor. These characters encode a path by specifying a series of out-ports. (Note that a token could never do such a thing, since a path in the network can grow arbitrarily long.)

We require two main snake types, which we call growing and dying⁵. *Growing snakes* are used to generate encoded paths of the network, and *dying snakes* are used to mark encoded paths. Our protocol requires two kinds of each of the two snake types; specifically, we will need out-growing, in-growing, out-dying, and in-dying snakes. “Out” and “in” are meant as a mnemonic; out-snakes are generated at the root and proceed outward from it, while in-snakes are generated elsewhere and trigger some action when they reach the root. Out-growing, in-growing, out-dying, and in-dying snakes will be referred to as OG-snakes, IG-snakes, OD-snakes, and ID-snakes in what follows.

Each of the four kinds of snake gets its own alphabet of characters to describe it; this allows processors to determine with which kind of snake they are dealing. We will spend a section on each type, elucidating its respective behavior. First, we need to go over some general rules common to all snake types.

3.3.1 General Snake-handling Rules.

- All snakes are speed-1.
- Snakes of different types do not interact. A processor can handle different snake types simultaneously without getting confused because snake

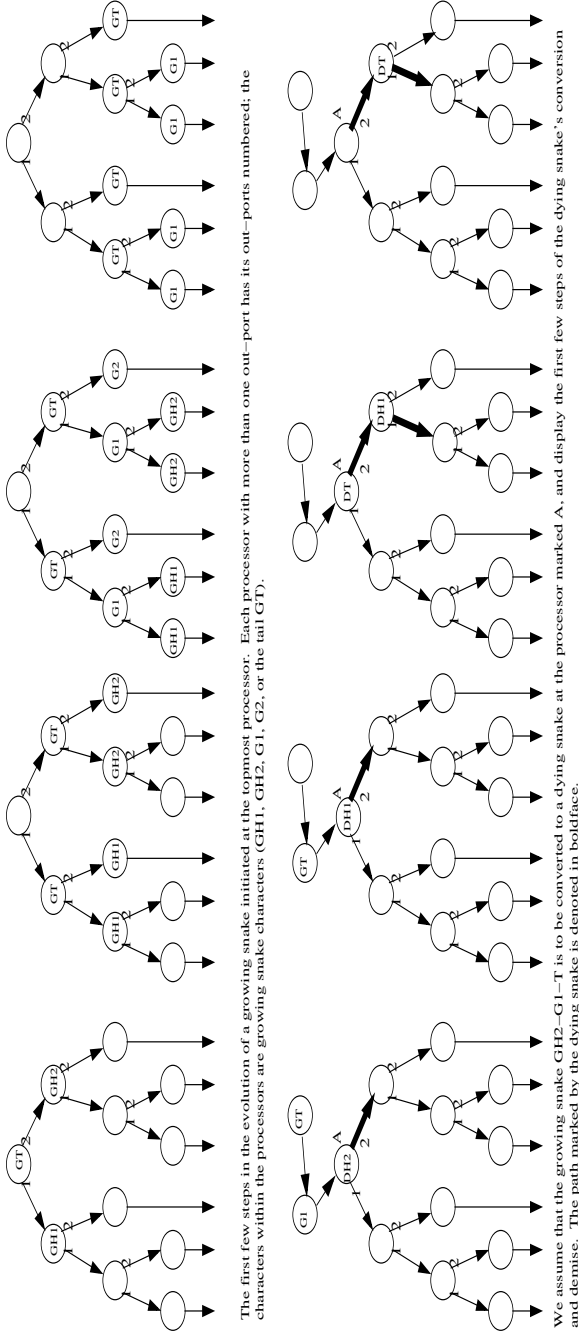
types are distinguished by their alphabets. Note that this does not impose arduous memory constraints upon the processors (which are finite-state machines) since there is only a constant number of snake types.

3.3.2 Growing Snakes. Growing snakes function as information generators. We define the *initiator* to be the processor from which the growing snakes first emanate. The *terminator* is defined to be the processor that the snakes are attempting to reach. Growing snakes grow in a breadth-first manner; the first growing snake to reach the terminator processor will have encoded within its body a minimal-length path from the initiator to the terminator. Upon reaching the terminator, a growing snake head might then initiate some further action based on the protocol and the state of the terminator. The rules for handling out-growing snakes are outlined below; the rules for handling in-growing snakes are identical (just replace “OG” with “IG”), except where noted.

- First, the head characters of the baby OG-snakes are generated by the initiator. This processor sends an OG-snake head character out of every out-port during the first time step. The particular head character to be sent will correspond to the out-port from which it is being sent. For every i between 1 and δ , the growing head snake character $OGH(i)$ will be sent through out-port i . During the next time step, the initiator will send a tail character OGT through every out-port. Thus a baby snake is born.
- When a processor receives an out-growing snake character for the first time, it marks itself *OG-visited*, and marks the in-port through which the growing snake character was passed as its *OG-parent*⁶. (These marks will be used later by certain breadth-first tokens; see Section 3.2.) Only this first OG-snake will be allowed to pass through the processor; all other OG-snake characters will be ignored. The OG-visited markings will never be removed; thus, an OG-snake will carve out a breadth-first-search tree. IG-snakes behave identically, except that IG-visited and IG-parent markings are removed from the network at one stage of our protocol, thus allowing a given processor to be visited by more than one IG-snake during its lifetime. Until this removal occurs, however, each

⁵Even, Litman, and Winkler in [6] refer to growing and dying snakes as adders and rattlers, respectively.

⁶If two or more OG-snakes arrive simultaneously, the one arriving through the lowest-numbered in-port is deemed “first.”



IG-snake will also carve out a breadth-first-search tree.

- When a processor receives a non-tail OG-snake character, it simply sends this character through all out-ports during the next time step. Once the processor sends the character out, it need not retain it in “memory.” (In this way, the processor simply passes the head and body through every out-port. Thus arbitrarily long paths can be generated.)
- Once a processor receives the tail of an OG-snake, instead of simply passing it through like the other body characters, the processor generates a new body character. For all i between 1 and δ , it simultaneously sends the character $OG(i)$ through out-port i ; thus a new body character is generated to mark the current processor's position in the path. Only after this new character is passed along does the processor send the tail through.

3.3.3 Dying Snakes. Dying snakes function as path markers. After a path is generated by the growing snakes, it is the responsibility of the dying snake to mark the generated path so that the processors on it will know (a) that they lie along a special path and (b) which in-port and out-port they should use for funneling information along the path. In our protocol, OD-snakes will be formed by converting the characters of an IG-snake into OD-snake characters as they pass through the root; ID-snakes will be created from OD-snakes in a manner to be described in Section 5. The rules for handling OD-snakes are outlined below; the rules for handling ID-snakes are identical (just replace “OD” with “ID”), except where noted.

- An OD-snake will mark a path generated by an OG-snake (see Section 5); thus, since an OG-snake carves out a breadth-first-search tree, the path will never self-intersect. Similarly, neither will a path that an ID-snake is to mark. However, the concatenation of the two paths (which, in our protocol, will always be a loop beginning and ending at the root) may self-intersect; any processor will appear at most twice on the concatenation. We will, eventually, want to consider the concatenation as a whole; to this end, we imbue each processor with two “predecessor in-ports” (numbered 1 and 2) and two “successor out-ports” (ditto).
- Whenever a processor receives the head character $ODH(i)$ of a OD-snake, it sets predecessor in-port #1 equal to the number of the in-port through which it received the character, and sets successor out-port #1 equal to i . These two values indicate

the two edges of the path incident to the processor. ID-snakes work identically, except that they use predecessor in-port #2 and successor out-port #2. The head character is then discarded (not sent through any out-port).

- If the next OD-snake character that the processor receives through the predecessor in-port is $OD(j)$, it gets sent through the successor out-port as $ODH(j)$. (In other words, the next OD-snake body character that comes through gets converted to the new head.) The processor then passes all further OD-snake characters received through its predecessor in-port to its successor out-port exactly as received. If the next character happens to be a tail, then it gets sent through the successor out-port as is⁷.
- An OD-snake only propagates along the path it is marking, and a maximum of one will be in the network at any given time, so we need not worry about OD-visited markings.
- If a processor receives only the tail of an OD-snake, it is an indication that the processor is special in some manner depending on the specifics of the protocol.

3.4 Marked loops. As mentioned in Section 3.3.3, we will be using dying snakes to mark certain loops (not necessarily simple) that begin and end at the root. We will refer to this structure repeatedly throughout the paper, and thus make the following definition:

DEFINITION 3.1. *A marked loop will be a loop marked by dying snakes in the manner described in Section 3.3.3. The root must be one of the processors on the loop. The loop may or may not be simple, but no processor or edge can appear more than twice on it.*

Each processor will have its predecessor and successor port (or, if necessary, ports) set by the dying snakes. A processor with only predecessor in-port #1 set will only accept a loop token through that in-port; it will subsequently pass the token through successor out-port #1⁸. Similarly, a processor with only predecessor

in-port #2 set will only accept a loop token through that in-port; it will subsequently pass the token through successor out-port #2. Finally, a processor with both predecessor in-ports set will initially accept a given loop token only through predecessor in-port #1 (it will pass the token through successor out-port #1, of course); it then waits for the token to come through predecessor in-port #2 (at which point it passes the token through successor out-port #2); it then will expect the next such loop token through predecessor in-port #1 again.

We will hereon refer to the predecessor in-port (resp. corresponding successor out-port) through which a loop processor awaits a loop token as the *appropriate* predecessor in-port (resp. successor out-port).

3.5 The Ring-of-Trees.

DEFINITION 3.2. *A ring-of-trees is a network consisting of a directed ring with trees hanging from the processors of the ring. The edges of the trees all point towards the leaf processors; note that, unless all of the trees are empty, the ring-of-trees is not strongly-connected. The root processor is one of the processors on the ring, and we require that the shortest path within the structure from the root processor to any leaf processor be less than the number of processors in the ring. This data structure and the following discussion were introduced in [7].*

A method for firing a directed ring of processors in $O(P)$ time (where P represents the number of processors on the ring itself) has been known for decades; we direct the reader's attention to the references mentioned in Section 2.2. It was shown in [7] that a ring-of-trees with P processors in its directed ring can be fired in precisely the same amount of time; we omit the fairly simple proof.

Now, if we can realize our network as a ring-of-trees that contains all of the network's processors, then we'll be able to fire it. How might this realization be done? Certainly no single finite-state processor can hold the entire structure of the ring-of-trees. However, this is not necessary; to pass information appropriately around the ring-of-trees, we need only that each processor know certain of its neighbors within the structure. Specifically, we will require that each processor on the directed ring portion of the ring-of-trees⁹ knows its predecessor in-port(s) and successor out-port(s), and that each tree processor knows its parent in-port. This

⁷In our protocol, OD-snakes will be converted into ID-snakes at those processors marked IG-start; this will provide an exception to these rules, for at those processors all OD-snake characters are converted into ID-snake characters instead. In addition, as might be expected, an IG-start-marked processor will set predecessor in-port #1 and successor out-port #2 appropriately as the dying snakes go through; its other two ports will not be needed, as we will show in Section 5.

⁸Once again, processors marked IG-start will provide an exception to this rule; they will accept a loop token only through

predecessor in-port #1, but will pass it through successor out-port #2;

⁹The ring, it should be noted, need not be simple, as long as no processor is included more than a fixed constant number of times; the arguments from [7] still work.

will make it easy to transfer information within the ring-of-trees structure: each processor on the ring can send information around it via the appropriate predecessor and successor ports, and can send information to the trees via all non-successor ports; each tree processor can simply send information through all out-ports. If each processor only accepts information through its appropriate predecessor or parent, the ring-of-trees structure will be realized.

So how do we find a directed ring within the network? We will use growing and dying snakes to mark a loop into our network (see Section 3.4 for the definition of a marked loop, and Section 5 for the specifics of how we use snakes to do this); this marked loop will serve as our directed ring. We claim that, given a marked loop passing through the root with length greater than Γ , we can realize the ring-of-trees within the network. To do so, we use the following procedure. Each processor on the marked loop already has a predecessor in-port (or in-ports). The root releases a copy of a TREE-PARENT token through each of its δ out-ports. This token acts as a cross between a breadth-first and a loop token; upon receiving it, a processor will perform one of the following actions, depending on its state.

- If the processor already has a parent designation and is not on the marked loop, the TREE-PARENT token is simply ignored.
- If the processor has no parent designation when the TREE-PARENT token arrives, the processor designates the in-port through which the token was received as its parent. It then passes a copy of the token through each of its out-ports.
- If the processor is on the marked loop (and hence has predecessor(s)), it checks whether the TREE-PARENT token came through its appropriate predecessor in-port; if so, it passes a copy of the token through its appropriate successor out-port and every non-successor out-port. If the token did not come through the appropriate predecessor in-port, the token is ignored.

Now, is the resulting structure a ring-of-trees? Unfortunately, not yet. For processors x and y , let $d'(x, y)$ be the distance from x to y within the structure; let L be the length of the marked ring. (Recall that we assume that $\Gamma < L$.) For the structure to be a ring-of-trees, we need $d'(\text{root}, y) < L$ for all leaves y in the network. This is not necessarily true. However, we can prove:

LEMMA 3.1. *For any leaf processor y , $d'(\text{root}, y) < 2L$*

Proof. Fix any leaf processor y and any ring processor r . If there exists a shortest path from r to

y that does not intersect the ring (other than at r), then by construction

$$d'(\text{root}, y) \leq d'(\text{root}, r) + d(r, y)$$

Consider a shortest path from root to y (not necessarily within the structure). Such a path must have length less than or equal to Γ . Let the final ring node that this shortest path intersects be final . By the above observation, we have

$$d'(\text{root}, y) \leq d'(\text{root}, \text{final}) + d(\text{final}, y)$$

If $\text{final} = \text{root}$, then the expression is less than or equal to Γ and hence less than $2L$. If $\text{final} \neq \text{root}$, then $d'(\text{root}, \text{final}) \leq L$ and

$$d(\text{final}, y) \leq d(\text{root}, y) - 1 < \Gamma \leq L \Rightarrow$$

$$d'(\text{root}, y) \leq d'(\text{root}, \text{final}) + d(\text{final}, y) < 2L$$

□

How can we use this fact? First, we note that once a token has traveled around the ring twice, the root can rest assured that the TREE-PARENT tokens have had opportunity to reach every processor in the network; hence every processor has a parent designation, as required. Second, we can use this fact to create a ring-of-trees from the above structure.

To do so, we "split" each of the ring processors into two distinct personalities – say, personalities "0" and "1". The root accomplishes this by sending a SPLIT token around the loop simultaneously with the TREE-PARENT token. All messages from processors on the loop have the number of the personality sending it attached. If a non-root processor n receives a message from personality i of its neighbor, then personality i of processor n handles it; if the root receives a message from personality i of its neighbor, then personality $i + 1 \bmod 2$ handles it. If we decree that every "tree" hanging from a ring processor in our structure hangs from personality 0, then we can simulate a twofold increase in the length of our marked loop, which is all we need for a ring-of-trees.

The astute reader will note that there is nothing special about the number "2" in the above argument; any fixed constant k would do in its stead. We thus have

LEMMA 3.2. *If we can mark a loop of length $\Theta(\Gamma)$, then we can fire the whole network; indeed, once we have such a marked loop, it takes only time $O(\Gamma)$ to fire it.*

Proof. Given a marked loop of size $\Theta(\Gamma)$, we can split it in time $O(\Gamma)$, form a ring-of-trees structure in

time $O(\Gamma)$, then fire it (using the protocol in [7]) in time $O(\Gamma)$. \square

It remains only to show that we can construct such a marked loop quickly; this will be the main thrust of our protocol (see Section 5).

4 Overview of the Protocol

In this section, we assume that we are given a Wake Up and Report Protocol that runs in time W , where W is dependent on the network topology and size. From this protocol, we will construct a Firing Squad Synchronization Protocol that runs in time CW , where C is a constant *independent* of network topology and size. Because a protocol for the FSSP is necessarily a protocol for the WURP¹⁰, this will be enough to prove the asymptotic time-equivalence of the two problems.

Before proceeding to the detailed steps of the constructed protocol for the FSSP, we will present a top-down version which illustrates the basic ideas. The given protocol for the WURP is used as a stopwatch. Since, by lemma 2.1, its running time W is $\Omega(\max(\Gamma, \Gamma'))$, we know that sufficiently many repetitions of the WURP will provide enough time for a message to travel from the root to any given processor (or vice-versa); this will be crucial in establishing the running time of our protocol.

While repeating the WURP, we build and dismantle increasingly longer loops through the root; each successive loop will be at least 4 times the length of its predecessor. After sufficiently many iterations of the WURP – i.e., in time $O(W)$ – we are guaranteed (for reasons to be explained in Section 6) that the length of the current loop is at least $\frac{1}{4}$. By lemma 3.2, once we have such a loop, we can fire it in time $O(\Gamma)$; since W is $\Omega(\Gamma)$, this suffices to prove the main result of this paper.

We will now elaborate on some of the more important steps in the brief description above. The process of constructing and dismantling increasingly longer loops in the network is accomplished via the use of various types of snakes. Growing snakes are sent out, and return carrying the encoded paths that allow the construction of the loops. (The steps for this are outlined more fully below.) When construction of a new loop begins, the previous loop is dismantled; when construction completes, we wipe out all incipient loops within a certain radius of the root, thus guaranteeing that the next loop, if any, will have length at least 4 times that of the current one. When the WURP “stopwatch” calls time, we finish whatever loop we are working on, if any, assured

that the last loop we mark will be extremely long; indeed, as mentioned above, its length will be at least $\frac{\Gamma}{4}$.

5 The Protocol (detailed description)

Recall that we are given a protocol for the Wake Up and Report Problem. From this protocol, we will construct a protocol for the Firing Squad Synchronization Problem which runs in no more than a constant multiple of the time-complexity of the given WURP protocol.

1. The root processor is signaled to commence the FSSP protocol. At this time it does two things simultaneously. First, the root commences the given WURP protocol for the first of “sufficiently many” times¹¹ (when it finishes, the root immediately starts it again). Simultaneously, the root begins the protocol described in steps 2 through 8 below¹².

STEPS 2 THROUGH 6 (EXCEPT 5) WILL BE DEVOTED TO MARKING A LOOP THROUGH THE NETWORK THAT INCLUDES THE ROOT. STEP 5 IS DEVOTED TO CLEARING THE EXTRANEIOUS SNAKE MARKINGS CREATED DURING THIS PROCESS.

2. The root releases out-growing snakes. Any processor receiving an out-growing snake character for the first time marks itself as “OG-visited”, thus preventing any subsequent OG-snakes from entering it. It will also designate the in-port from which it received the OG-snake as its “OG-parent” in-port. The “OG-visited” mark will never be cleared; hence, the OG-snakes carve out a breadth-first-search tree.
3. A processor that receives an OG-snake will also create a corresponding in-growing snake, unless an already-created IG-snake accompanies the OG-snake, in which case the processor will pass the latter along, in accordance with growing-snake rules. In either case, any OG-snake character broadcast by the processor will be accompanied by the corresponding IG-snake character; e.g., if the processor broadcasts the character OG(2), it also must broadcast the character IG(2).

¹¹For those who crave explicitness, we will show later that “sufficiently many” is no more than 8. It may well be possible to reduce this number by making the following protocol more efficient; an improvement of this sort will not affect our final asymptotic result, however.

¹²We assume that the two protocols do not interfere with each other; running two protocols at once merely requires us to enlarge the language and state space by a constant factor, since each requires only a constant amount of language and state space.

¹⁰Clearly, a processor must be non-quiescent in order to fire. Hence, upon firing, the root can be certain that there are no quiescent processors in the network.

In-growing snakes do not respect the OG-visited and OG-parent markings created by OG-snakes; instead, they leave analogous “IG-visited” and “IG-parent” markings. In addition, any node that creates an IG-snake marks itself “IG-start.” Unlike the OG-markings, however, these IG-markings may later be deleted, along with the snakes that made them (see step 5).

We note for clarity that, once created, an IG-snake head will accompany any given OG-snake head wherever the latter goes, until such time as the IG-snake is deleted. Thus, until that deletion occurs, no new IG-snakes will be created. For example, when the protocol first begins, IG-snakes will be created at all processors whose distance from the root is one, and nowhere else until the IG-snakes are deleted. Once this deletion occurs, new IG-snakes will be created en masse if and when the OG-snakes reach as-yet-unvisited processors. Until its deletion begins, a generation of IG-snakes will carve out a forest of breadth-first-search trees.

4. When the root receives the head of an IG-snake, it does three things. First, it closes itself off to all other IG-snakes, ignoring any that attempt to enter (this closure will continue until step 6, even if the root’s IG-visited mark is cleared in the interim). Second, if a loop has already been marked within the network¹³, the root begins the process of unmarking it; it does so by sending the speed-1 loop token UNMARK around the loop. Each processor in the old marked loop, upon receiving the token through its appropriate predecessor in-port, passes the token through the appropriate successor out-port, then forgets those predecessor and successor designations¹⁴. Third, we note that the characters of the incoming IG-snake encode a path from the root back to itself; this will be our new loop. The root then converts this IG-snake into the corresponding OD-snake (i.e., the root must therefore eat the head character of the IG-snake as if it were an OD-snake character, then send the rest of the snake through the appropriate

out-port, converting each IG-snake character into the corresponding OD-snake character.) The OD-snake will mark that part of the loop generated by the OG-snake; when the OD-snake reaches a processor marked IG-start, the process converts it an ID-snake, which proceeds to mark that part of the loop generated by the IG-snake.¹⁵ The ID-snake will die as it returns to the root; that is, the root will receive just the tail of the ID-snake.

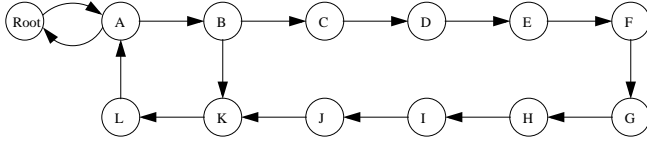
5. When the marking of the new loop is complete (this occurs when the tail of the ID-snake reaches the root), the root does two things. First, it sends the speed-1 loop token CLOCK around the new loop. This token does nothing except get passed from processor to processor along the marked loop until it returns to the root. Second, we now want to remove the IG-snake characters and markings still percolating through the network. To this end, the root broadcasts breadth-first speed-4 PREPARE-TO-KILL tokens that propagate along the OG-tree. (Any processor receiving a PREPARE-TO-KILL token that does not come from its OG-parent ignores the token.) The PREPARE-TO-KILL tokens will simultaneously reach the processors that have an IG-start marking. (This will be proven later.) When this occurs, they change into breadth-first speed-4 KILL tokens that propagate along the IG-tree, deleting IG-snake characters and clearing IG-start, IG-visited, and IG-parent markings as they go¹⁶.
6. The CLOCK token will return to the root at the same moment that the KILL tokens finish deleting all IG-snakes and markings from the network.

¹³If no loop has yet been marked, ignore this step.

¹⁴We note here that, although the unmarking of the old loop will be complete before we finish marking the new loop, the loops may still interfere. However, we will never have more than one loop in the network at a time. Each processor must be able to keep track of two loops, locally (i.e., which edges are incident to each loop.) The root (which is in every loop) must keep track of a parity bit to distinguish between loops; it should send this out with every dying-snake character or loop token so that processors on the loop can tell which loop it affects. This is an annoying but unimportant complication, and we will say no more about it.

¹⁵We convert OD-snakes to ID-snakes so as to avoid complications when that part of the loop generated by the OG-snake intersects the part generated by the IG-snake; see Sections 3.3.3 and 3.4 for loop-marking details. Notice that the processor marked IG-start cannot appear twice on the loop, since it ends the first part and starts the second, and neither part is self-intersecting; this justifies an assertion made in Section 3.3.3.

¹⁶This seems to be a great deal of effort, and one might wonder why such effort is justified. Note that if we are not on our first marked loop, then there must have been at least one generation of IG-snakes that has been deleted by KILL tokens. Many processors – including, perhaps those near the out-ports of the root – may have had their IG-visited markings cleared at that time and never replaced by a subsequent IG-snake. If we simply allow the root to release KILL tokens, they may run into these non-IG-visited processors and be ignored. Thus we arrange matters so the KILL tokens will be activated at those processors with IG-start markings; these processors serve as the bases for the trees in the IG breadth-first forest, so are a convenient place to start the deletion of IG-snakes and associated markings.



The first loop to be marked will be the smallest one, Root–A–Root, which has a length of 2. After the marking of this loop has been completed, all IG–snakes and marks will be eliminated from the network; this process will finish when the OG–snakes are at processors whose distance from the root is precisely 8 (=4 times 2). A new IG–snakes will be created when the OG–snakes move on to those heretofore quiescent processors that are at a distance of 9 from the root (in the network above, the only such processor is I). The upshot of this: the loop Root–A–B–K–L–A–Root will not be marked, as all its processors are within 8 of the root; the second marked loop will be Root–A–B–C–D–E–F–G–H–I–J–K–L–A–Root.

Figure 2: The main idea of step 6

At this point, the root opens itself up to IG–snakes once more. Let L be the length of the just-completed loop. If there are still quiescent processors in the network, then we make two claims: first, the distance from the root to the most distant non-quiescent processor is exactly $4L$; second, a new generation of IG–snakes will be simultaneously spawned when the OG–snakes subsequently reach distance $4L + 1$ from the root. For a proof of these facts, see Lemma 6.1; for a visualization of this step, see Figure 2.

7. The root repeats steps 4 through 6 until the given WURP protocol has run sufficiently many times (where what “sufficiently many” is will be determined later). At that time, the root completes whatever loop, if any, it is working on, and eliminates all IG–snakes and markings remaining in the network.
8. At this point, we have a single marked loop in the network. This loop, we will show, must have length at least $\frac{L}{4}$. As described in Section 3.5, we can now split this loop fivefold, create a ring-of-trees from the lengthened loop, and then fire the ring-of-trees.

6 Proof of the Protocol’s Correctness

6.1 Lemmas.

LEMMA 6.1. *Let L be the length of the current marked loop (if no loop has been marked yet, let L be zero.) We claim the following loop invariants:*

1. *Let T be the number of time ticks elapsed since the start of our protocol. At $T = 3(4L)$, the marking of this loop was completed. At $T = 4(4L)$, all IG–snakes were eliminated from the network, along with all IG–visited, IG–start, and IG–parent markings; one time tick later, all KILL tokens were gone.*
2. *The next generation of IG–snakes (if any) will all be created at $T = 4(4L + 1)$, at all processors whose*

distance from the root is $(4L + 1)$; these processors will all be marked IG–start, of course.

3. *The next loop (if any) will have length greater than $4L$.*

Proof. We proceed by induction on the number of loops. All claims are trivial when $L = 0$. Now, how much time had elapsed when we finished marking the current loop? Well, $4L$ had passed when the head of the IG–snake that encoded the loop reached the root¹⁷; $4(3L)$ had passed by the time the tail of the IG–snake reached the root, indicating the end of the loop-marking process. At this point, the speed-4 PREPARE-TO-KILL were released; we may assume (by the inductive hypothesis) that they simultaneously became speed-4 KILL tokens when they reached the processors marked IG–start. These KILL tokens caught up to the IG–snake heads farthest from the root at $T = 4(4L)$. Were they able to eliminate all IG–snakes and markings en route? Certainly; as noted in Section 5, before the deletion process begins, a generation of IG–snakes carves out a forest of breadth-first-search trees anchored at those processors marked IG–start. Our KILL tokens began working at those processors and raced their way up the trees, annihilating all IG–snakes and markings¹⁸. One time tick after $T = 4(4L)$, before the OG–snake heads could move on, each of the KILL tokens fell off its associated IG–snake breadth-first-search tree, so died.

Now, at $T = 4(4L + 1)$, the OG–snake heads move on, reaching all processors that are a distance of $4L + 1$ from the root. Since the OG–snake heads are unaccompanied by IG–snake heads, the processors simultaneously make a new generation IG–snakes (and mark themselves IG–start.) As pointed out in step 3 of the protocol in Section 5, no more will be created until the generation is eliminated.

Finally, the first IG–snake from this generation that reaches the root will form the next loop; this loop will be the shortest one in the network that contains a processor that is at least $4L + 1$ from the root. In any case, however, the length of the loop will clearly be more than $4L$.

¹⁷The ubiquitous 4’s in these calculations (well, most of them) are because snakes travel at speed 1.

¹⁸It’s possible for an IG–snake to “double-back” to previously-visited processors once the deletion process has begun; nonetheless, the region between a KILL token and the heads of its associated IG–snakes will always be a tree, so the KILL token will have no difficulty eliminating all IG–snakes and markings, even those snakes that double-back. We point out that this lack of difficulty is a large advantage of activating the KILL tokens at those processors marked IG–start.

LEMMA 6.2. *The root will receive at least one IG-snake (and hence start the creation of a marked loop) within time $4(\Gamma' + 1)$ of the beginning of the protocol.*

Proof. First, recall that all snakes are speed-1 and thus move along edges every 4 clock ticks. After 4 clock steps, at least one IG-snake – call it s – is released. Now, either snake s makes it back to the root in at most $4\Gamma'$ more clock ticks, or it is killed along the way. However, since KILL tokens are only released from the root after the root has received an IG-snake, the only way IG-snake s can be killed is by running into an IG-snake that is no farther away from the root than itself. But since a collision of two snakes always results in the survival of one, at least one IG-snake among those no farther away from the root than s must survive to reach the root.

LEMMA 6.3. *The "sufficient number" of WURP iterations mentioned in Step 7 of our protocol is 8. After WURP has been run eight times, the root has completed whatever loop (if any) it is working on, and all IG-snakes and markings have been eliminated from the network, we can rest assured of two things: first, at least one marked loop will have been created; second, the OG-snakes will have visited every processor in the network.*

Proof. By Lemma 2.1, any WURP protocol takes time greater than $\max(\Gamma, \Gamma')$; thus 8 iterations must take at least time $T = 4(\Gamma + \Gamma')$. Note that $T \geq 4(\Gamma' + 1)$; thus, by Lemma 6.2, at least one marked loop will have been created. Note also that $T \geq 4(\Gamma + 1)$, so we know that the OG-snakes will have visited every processor in the network.

When step 7 is completed, we will have a unique last marked loop in the network. Let M be its length.

LEMMA 6.4. $4M \geq \Gamma$

Proof. Let T be the number of time ticks since the start of the protocol. When the last loop is completed and all the IG-snakes and markings are cleaned up, we have $T/4 = 4M$ (by the same logic as in the proof of Lemma 6.1).

We must also have $T/4 \geq \Gamma$ at this point, or else there would be non-OG-visited processors in the network, which would contradict Lemma 6.3. We therefore have $4M = T/4 \geq \Gamma$, which is the result we desire.

So we have a loop of length at least $\frac{\Gamma}{4}$, which is $\Omega(\Gamma)$; we are done.

6.2 Proof of the Protocol's Running Time.

THEOREM 6.1. *Given any WURP protocol (with running time W), the protocol presented in Section 5 creates a FSSP protocol which runs in time $O(W)$.*

Proof. Fix a network and WURP protocol; let W be its running time. We proved in Lemma 2.1 that W is $\Omega(\max(\Gamma, \Gamma'))$, so any $O(\Gamma)$ process is certainly $O(W)$. The main loop of the protocol takes $8W$ time (recall that we found 8 iterations of the WURP sufficient); marking the final loop is certainly an $O(W)$ process, as is splitting it; finally, firing the network takes $O(W)$ time. Thus, the protocol takes $O(W)$ time, as claimed.

7 Asymptotic Time-Equivalence of Other Problems

We have shown that the Firing Squad Synchronization Problem and the Wake Up and Report Problem are asymptotically time-equivalent. Other common network protocols can be easily shown to be asymptotically running-time equivalent to these. In this section, we illustrate several immediate examples. We specify that in each problem below, the root must enter a specific terminal state before the protocol can be considered completed. Thus, it is implicit in all of the following protocols that the root must be informed when the network has completed its assigned task before the protocol terminates.

7.1 Build a Rooted Outgoing Spanning Tree.

The tree created by the out-growing snakes in our Firing Squad Synchronization Protocol can be utilized as a rooted outgoing spanning tree. Conversely, if all processors must have parent designations, they must have awakened at some point to get their designations; thus the protocol for building a rooted spanning tree can also be used as a Wake Up and Report Protocol.

7.2 Long Circuit/Slow Clock.

The Long Circuit/Slow Clock problem asks for a rooted outgoing spanning tree and a loop through the network such that the length of the loop is at least a constant multiple of the length of the longest branch of the tree. The loop created in our Firing Squad Synchronization Protocol can be used as a Long Circuit/Slow Clock; conversely, given a Long Circuit/Slow Clock, one can easily construct a Wake Up and Report Protocol (simply send a token sufficiently slowly around the circuit while broadcasting a wake-up message so that one can be sure that every processor has awakened by the time the token returns.)

7.3 Conducting a Network Search. The objective of conducting a Network Search is for the root to query

the network for some piece of “information.” This information can be stored in the form of bits in one or more processors. The root terminates the protocol when either (a) it gets notification that the information it requires exists in some specific processor in the network and creates a marked path to said processor or (b) it finds that the information it requires does not exist anywhere in the network.

One possible protocol which solves the Network Search Problem is as follows: Have the root broadcast a breadth-first query token to the network. Simultaneously, have the root initiate a WURP protocol. The protocol for each of the processors in the network that contain the required bits of information will be to simply broadcast IG-snakes which follow exactly the same rules as outlined in Section 5 above. Otherwise, simply propagate the root’s query message.

If the root does not receive the head of a IG-snake by the time 8 WURP protocols have been executed, the root can terminate the protocol with the knowledge that the information it requires does not exist anywhere in the network. We know that any WURP protocol must take longer than $\max(\Gamma, \Gamma')$. Thus 8 executions must take time greater than $4(\Gamma + \Gamma')$, which is an upper bound for the amount of time that it would take for the query to reach any given processor and have any potential IG-snake return.

If the root does happen to receive the head of a IG-snake before 8 WURP protocols have been executed, it simply establishes the marked path as illustrated in Section 5.

In either case, the running time is at most a constant multiple of the given WURP protocol’s running time.

Clearly, if the information is not to be found in the network, a Network Search must query every processor. Thus, at termination, every processor must have been awakened. We have therefore established the required equivalence.

Acknowledgments

The authors would like to thank the various referees, who pointed out several protocol simplifications and presentation suggestions.

References

- [1] Yehuda Afek and Eli Gafni. Distributed algorithms for unidirectional networks. *SIAM Journal on Computing*, 23(6):1152–1178, 1994.
- [2] D. Angluin. Local and global properties in networks of processors. In *Proceedings of the 12th ACM Symposium on Theory of Computing (1980)*, 82–93, 1980.
- [3] Baruch Awerbuch and Rafail Ostrovsky. Memory-efficient and self-stabilizing network RESET (extended abstract). In *Symposium on Principles of Distributed Computing*, pages 254–263, 1994.
- [4] Ernest J. H. Chang. Echo algorithms: Depth parallel operations on general graphs. *Software Engineering*, 8(4):391–401, 1982.
- [5] B. Codenotti, P. Gemmell, and J. Simon. Symmetry breaking in anonymous networks. *manuscript (1995): citeseer.nj.nec.com/codenotti95symmetry.html*, 1995.
- [6] S. Even, A. Litman, and P. Winkler. Computing with snakes in directed networks of automata. *J. Algorithms*, 24(1):158–170, 1997.
- [7] N. Honda and Y. Nishitani. The firing squad synchronization problem for graphs. *Theoretical Computer Science*, 14(1):39–61, 1981.
- [8] K. Kobayashi. The firing squad synchronization problems for a class of polyautomata networks. *J. Comput. System Sci.*, 17(3):300–318, 1978.
- [9] A Mayer, Y Ofek, R Ostrovsky, and M Yung. Self-stabilizing symmetry breaking in constant-space. In *STOC92 Proceedings of the 24th Annual ACM Symposium on Theory of Computing*, pages 667–678, 1992.
- [10] J. Mazoyer. An overview of the firing synchronization problem. In *Automata Networks, LITP Spring School on Theoretical Computer Science, Angèlès-Village, France, May 12-16, 1986, Proceedings*, volume 316 of *Lecture Notes in Computer Science*. Springer, 1988.
- [11] M. Minsky. *Computation: Finite and Infinite Machines*. Prentice-Hall, Englewood Cliffs, NJ, 1967.
- [12] E. F. Moore. *Sequential Machines, Selected Papers*. Addison Wesley, Reading, MA, 1962.
- [13] R. Ostrovsky and D. Wilkerson. Faster computation on directed networks of automata. In *Proceedings of the Fourteenth Annual ACM Symposium on Principles of Distributed Computing*, pages 38–46, Ottawa, Ontario, Canada, 2–23 August 1995.
- [14] A. Segall. Distributed network protocols. *IEEE Transactions on Information Theory*, IT-29(2):23–35, 1983.
- [15] G. Singh. Efficient distributed algorithms for leader election in complete networks. In *Proceedings of the 11th International Conference on Distributed Computing Systems (ICDCS)*, pages 472–479, Washington, DC, 1991. IEEE Computer Society.