Directed network protocols

G. Tel

RUU–CS–87–6
February 1987
Revised October 1987

Rijksuniversiteit Utrecht

Vakgroep informatica

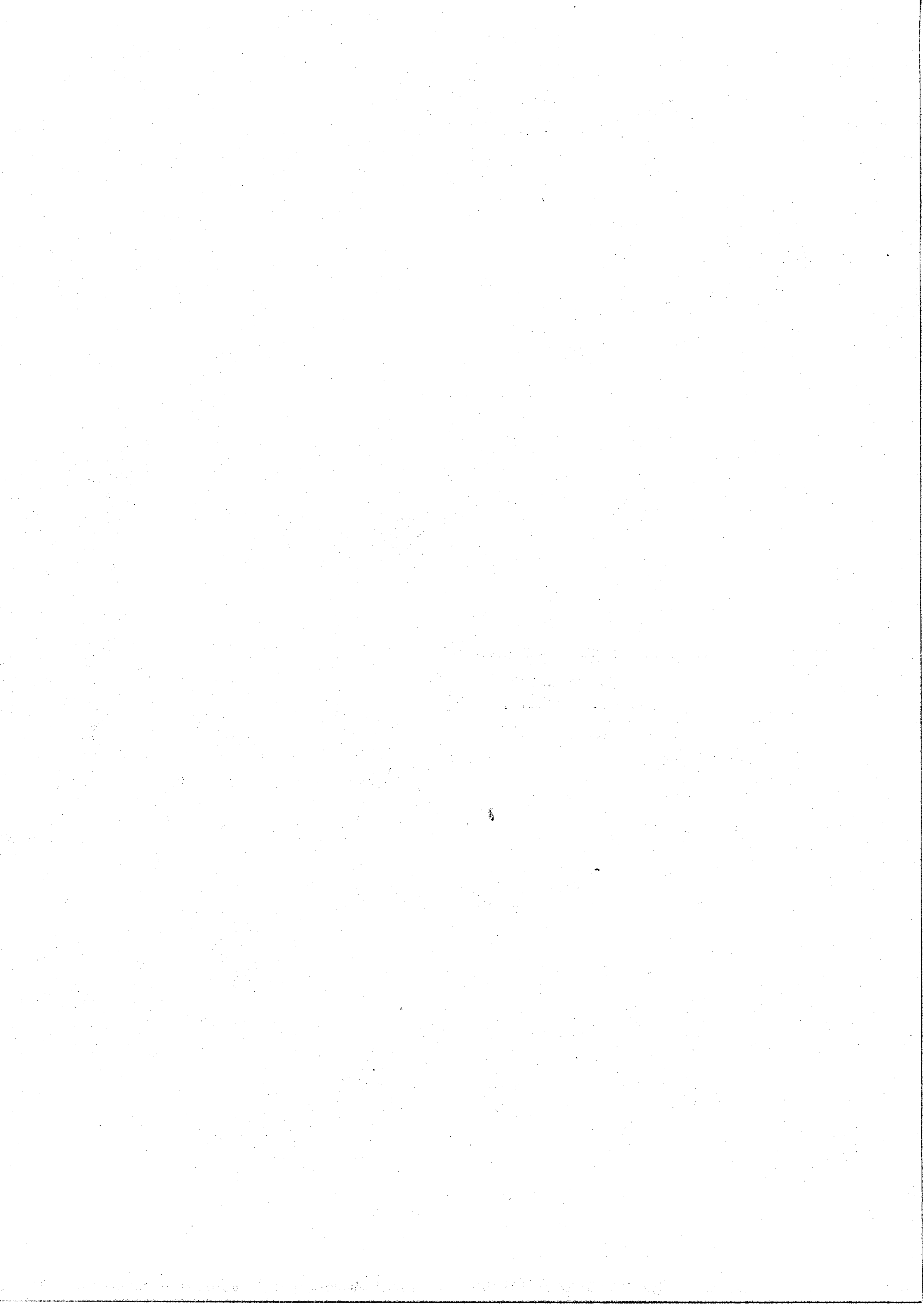
Budapestlaan 6 3584 CD Utrecht
Corr. adres: Postbus 80.012 3508 TA Utrecht
Telefoon 030-53 1454
The Netherlands

Directed network protocols

G. Tel

Technical Report RUU–CS–87–6
February 1987
Revised October 1987

Department of Computer Science
University of Utrecht
P.O. Box 80.012, 3508 TA Utrecht
The Netherlands



Directed network protocols

Gerard Tel

Department of Computer Science, University of Utrecht,
P.O. Box 80.012, 3508 TA Utrecht, The Netherlands.

Abstract: We present a uniform approach to the description and validation of several known and unknown distributed algorithms for solving control problems in computer networks using unidirectional communication. After introducing two basic protocols, we use these to build algorithms for Resynch [Fi79], Connectivity, Min Hop Routing and Distributed Infimum Approximation [Te86]. All protocols are extended in a uniform way to cope with changing network topology and the failure of nodes and links. The protocols are all optimal in time complexity.

1 Introduction

Segall [Se83] has presented a uniform approach to the development and validation of several distributed algorithms for solving network problems. The protocols Segall develops use that communication channels in his model of a network are bidirectional. In this paper we will give a similar approach to the development of protocols, suitable for networks employing *unidirectional* communication. In most of our protocols we do not even need the "FIFO" assumption, that is, we allow messages sent over a link to arrive in a different order. We do assume only that messages arrive in finite time and unchanged.

Basic in the work of Segall are two protocols, for PI (Propagation of Information) and for PIF (Propagation of Information with Feedback). In the next section we will give protocols, satisfying the same specifications, that work on directed networks. Starting from there we will build protocols for many network problems, such as Resynch [Fi79], Election, Distributed Infimum Approximation [Te86] (thus covering Distributed Termination Detection [Fr80] also), Connectivity [Se83], and Minimum Hop Routing, all suited for directed networks. The correctness proofs of the algorithms are highly modular, as advocated by Gafni [Ga86]. This report can be seen as an attempt to prove the usefulness of Gafni's ideas for this particular class of networks.

This work was supported by the Foundation for Computer Science (SION) of the Netherlands Organization for the Advancement of Pure Research (ZWO). The author's UUCP address is: ..!mcvax!ruuinfvax!gerard.

In our model a directed network consists of a set $\mathbb{P}$ of processes, connected by directed communication channels. A process $p \in \mathbb{P}$ knows sets In_p and Out_p, the incoming and outgoing edges of p. The channels deliver messages in finite time and error free but (unless stated otherwise) not necessarily in the order sent. Unless stated otherwise we assume the network to be strongly connected. If there is a communication channel from p to q we say p is a *predecessor* of q and q is a *successor* of p. If there is a directed path from p to q we say p is an *ancestor* of q and q is a *descendant* of p. Under the strong connectivity assumption this is of course the case for all p and q.

2 The basic protocols

We start describing a protocol PI for the propagation of some information M. If a process p has knowledge of M and wants all processes to know M also, it starts the protocol spontaneously. Processes that start the protocol spontaneously are called *starters*. Process p has a boolean state variable $done_p$, with initial value *false*. All protocols are given for the process with identity p.

Protocol PI:

```
Spontaneously or upon receipt of M do:
    if not done_p then
        begin done_p := true ;
            forall j ∈ Out_p do send M over j
        end.
```

For this protocol PI we can prove:

Theorem 2.1: Within finite time after one or more processes start the algorithm spontaneously, all processes will learn M. All processes will send M exactly once over all outgoing channels.
Proof: Each process sends M at most once over each channel because *done* is set when the process sends, and **not** *done* is a guard for sending. If a process sends, all successors of this process will eventually receive M and send also, unless they sent earlier already. By induction it follows that all descendants of starters will send exactly once. □

Each process will receive exactly one copy of M over each incoming channel, and thereafter receive no more messages. Thus, memory space for state information ($done_p$) can be allocated on first receipt of M, and removed when a copy has been received over all channels. In the sequel we will abbreviate "**forall** $j \in Out_p$ **do** send M over j" to "**shout** M".

It may be the case that the processes in the network must be informed about when all the processes have learned M. This is what we call *feedback*. Segall obtains feedback by modifying PI in a sense similar to Dijkstra and Scholten's termination detection algorithm for diffusing computations [DS80]. Processes delay sending of a copy of M to their *father* (the process from which they received M for the first time) until copies of M have been received over all links. To do this, links must be able to carry messages in two directions and also there must be exactly one starter. The protocol we give here for unidirectional networks is symmetric.

For conciseness of description we assume that a message that is received is seen immediately by the receiving process, but not removed from the incoming channel message queue. This is done by an explicit **consume** command. Of course it is also possible to remove it from the channel message queue immediately, and store it in a separate queue. Process p keeps an integer state variable $phase_p$, with initial value 0. D is a constant, known to all processes. We assume D to be an upper bound for the largest distance between two processes in the network.

Protocol PIF:

Spontaneously or upon receipt of M do:
    if $phase_p = 0$ **then**
        begin $phase_p := 1$; **shout** M **end** ;
    if all incoming queues contain a message now **then**
        begin consume one message from each queue ;
            $phase_p := phase_p + 1$;
            if $phase_p \le D$ **then shout** M
                        else *ready*
        end.

For protocol PIF we have the following:

Lemma 2.1: Within finite time after some processes start the algorithm spontaneously, all processes will enter the protocol.

Proof: As a result of entering in the protocol (either spontaneously or upon receipt of M) a process will shout M. It follows as in theorem 2.1 that all processes will eventually do so. □

By $d(p,q)$ we denote the length of the shortest path leading from p to q. Note that by assumption $d(p,q) \le D$ for all p, q.

Lemma 2.2: When some q reaches the statement *ready* all $p \in \mathbb{P}$ know of M.

Proof: If an edge pq exists then $phase_q \leq phase_p + 1$ because q must have received a message from p each time $phase_q$ is incremented. It follows that $phase_q \leq phase_p + d(p,q) \leq phase_p + D$ in general. Hence if $phase_q > D$ then $phase_p > 0$ and it follows that p has entered the protocol already. □

Lemma 2.3: All $p \in \mathbb{P}$ will reach the statement *ready*.

Proof: All processes will shout a first time by lemma 2.1. Hence, all message queues will eventually contain a first message, thus enabling all processes to shout a second time, etc. Eventually all processes will consume a Dth time and execute *ready*. □

Theorem 2.2: Within finite time after some processes start the protocol spontaneously, all processes will be notified that all processes have received M.

Proof: Lemmas 2.3 and 2.2. □

Finally, note that each process shouts and consumes D times. Hence, no messages remain in the system after the protocol has terminated in each node. Thus, memory space for state information ($phase_p$) can be allocated on first receipt of M, and removed after the Dth consume.

D must be an upper bound for the diameter of the network, i.e., $D \geq \max_{p,q} d(p,q)$. D can be loaded at system startup time or computed at runtime (see [SSP85] and section 7). Denote by D_p the *relative diameter* of p, i.e. the value $D_p = \max_q d(q,p)$. The protocol would also work if p stopped after D_p phases, although an extra shout would then be necessary to avoid deadlock. Here the property mentioned after theorem 2.2 no longer holds.

The message complexity of the PIF protocol is quite high. As usual we denote by N the number of processes and by E the number of communication channels. The PI protocol uses one message on each channel (theorem 2.1) and thus needs E messages, which is proven to be optimal by Gafni et al. [Ga84]. The PIF protocol sends D messages on each channel and thus costs $D \bullet E$ messages. Obviously $D \bullet E$ is bounded above by $O(N^3)$, a simple example graph (see figure 1) shows it is also $\Omega(N^3)$. The bit complexity of PIF (i.e., the total number

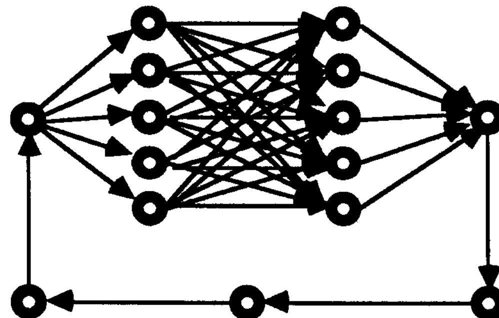


Figure 1

of bits communicated by the algorithm) can be reduced by shouting M only in the first round, and sending empty frames in later rounds. If M consists of m bits, this reduces the bit complexity from $D \cdot E \cdot m$ to $E \cdot m + D \cdot E$. The time complexity of PIF and all derived protocols is $O(D)$: within time D from the first spontaneous start all processes participate in the algorithm, and from then on each phase takes time 1. Obviously, a time complexity of $O(D)$ is optimal for a network of diameter D.

We have assumed strong connectivity of the network. The protocol will not work correctly on weakly connected networks. If a strongly connected component, containing starters, has descendants that are not ancestors, the algorithm may terminate in the starters' component before the descendants have received the message. If a strongly connected component, containing starters, has ancestors that are not descendants, these ancestors may never wake up, thus containing deadlock in the starters' component.

In fact, algorithms for weakly connected graphs do not exist. Consider a strongly connected graph G, containing a node x with one incoming and one outgoing edge (see figure 2). Before any node (other than x) can get feedback, x must send a message to y_2 confirming the receipt of M. Now replace x by two nodes x_1 and x_2 as in figure 2. Unless x_2 is a starter it will never send a message to y_2. The nodes other than x, x_1 or x_2 can never determine whether they are in G or in G'. It follows that any algorithm that works correct in G will deadlock in G'.

One can "decompose" PIF into an algorithm for synchronous networks and a so-called *synchronizer* [Aw85]. Assume the network operates synchronously, each message takes time 1 to arrive and internal computation takes time 0. Then the following would work:

Spontaneously or upon receipt of M:
    if not $done_p$ **then**
        begin $done$:= $true$; **shout** M ; $timer$:= D **end.**

The timer is decremented each time unit and when it reaches 0, every process must have received M. We get our earlier protocol by superimposing on this code a simplified version of Awerbuch's α-synchronizer [Aw85]. Awerbuch also gives synchronizers with a lower

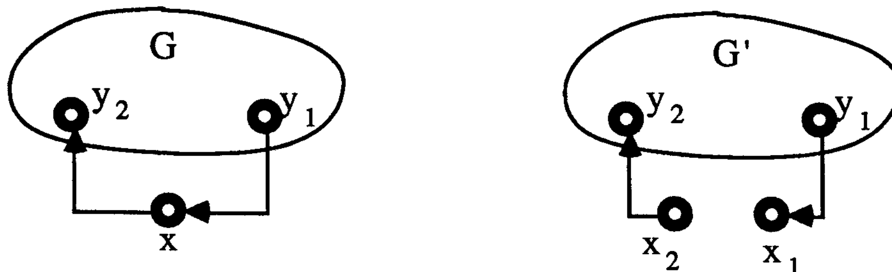


Figure 2

message complexity, and using these the time complexity of PIF could be decreased also. However, the time complexity of these synchronizers is higher and they are not as symmetric as our protocol. Furthermore, their use requires the existence of known special subtopologies, which we do not want here.

There are alternatives for the PIF protocol given here. Gafni and Afek [GA84] give an algorithm that has a lower message complexity and does not require knowledge of D. However, their algorithm is complex, not symmetric and has a high time complexity. Kutten [Ku87] gives a traversal algorithm using which a token, starting in one node, traverses a directed network an returns in the originator. Its message complexity is even lower than the complexity of the Gafni and Afek algorithm, but it has the extra drawback that there must be exactly one starter. This results in the needs for an election algorithm in some cases. Both algorithms can not be used in some of our applications.

3 Resynch protocols

The resynch problem, as formulated by Finn [Fi79] asks to bring all processes in $\mathbb{P}$ in a special state *synch* and thereafter in a state *normal*, in such a way that all processes are in *synch* before any of them goes back to *normal*. Finn [Fi79] and Segall [Se83] give complex algorithms for the resynch problem. We argue [TT87], that a resynch protocol can be easily obtained from any protocol for the PIF problem. Take such an algorithm, and modify it as follows: as soon as q receives M for the first time (or spontaneously if q is a starter) q sets its state to *synch*. A process getting feedback (i.e., receiving enough information to conclude that every process has received M) goes back to *normal*. The PIF protocol given earlier thus transforms to:

Protocol RES:

```
Spontaneously or upon receipt of M do:
    if phase_p = 0 then
        begin state_p := synch ;
            phase_p := 1 ; shout M    end ;
    if all incoming queues contain a message now then
        begin consume one message from each queue ;
            phase_p := phase_p + 1 ;
            if phase_p ≤ D then shout M
                        else state_p := normal
        end.
```

Theorem 3.1: Within finite time after some processes start RES spontaneously, all processes will be in state *synch*. Within finite time thereafter all processes will be in state *normal.*

Proof: Follows from lemma's 2.3 and 2.2. □

For the message M we can take a fixed, small message of size $O(1)$ bits, so the bit complexity of the protocol equals its message complexity, $O(D \bullet E)$. This is considerably less than the protocols in [Fi79] and [Se83].

4 Infimum computation

We now consider the following problem: given are a partially ordered set $(X, \leq)$ and a value $r_p \in X$ in each process $p \in \mathbb{P}$. We want to compute $I = \inf\{r_p : p \in \mathbb{P}\}$. We argue [TT87], that a protocol for the infimum problem can be easily derived from any protocol for the PIF problem: each p maintains a variable i_p, containing the infimum of all X-values p has seen so far and attaches this value to every message it sends for the PIF protocol. A process p can get feedback only if for all $q \in \mathbb{P}$ there is a chain of messages leading from q to p. (If there is not such a chain another execution of the PIF protocol can be constructed, in which p also gets feedback, but some q did not receive M yet, a contradiction.) It follows that at that moment $i_p \leq r_q$ and hence $i_p \leq I$. $i_p \geq I$ can be obtained easily. The PIF protocol from section 2 thus transforms to:

Protocol INF:

Spontaneously or upon receipt of $<i>$ do:
    if $phase_p = 0$ **then**
        begin $i_p := r_p$; $phase_p := 1$; **shout** $<i_p>$ **end** ;
    if all incoming queues contain a message now **then**
        begin consume one $<i>$ from each queue ;
            $i_p := \inf\{i_p, i$ read in previous statement$\}$;
            $phase_p := phase_p + 1$;
            if $phase_p \leq D$ **then shout** $<i_p>$
                        else *ready*
        end.

Lemma 4.1: Within finite time after some processes start the algorithm spontaneously, all processes will execute *ready* and no messages remain in the system.

Proof: Theorem 2.2. □

By $i_p^{(k)}$ we denote the value, computed by p while $phase_p = k$, i.e., $i_p^{(0)} = r_p$, and for $k > 0$, $i_p^{(k)}$ is computed after the kth consume.

Lemma 4.2: $i_p^{(k)} \leq r_q$ for $k \geq d(q,p)$.

Proof: By induction on $d(q,p)$. Note that $i_p^{(k')} \leq i_p^{(k)}$ for $k' \geq k$ so it suffices to show that $i_p^{(d(q,p))} \leq r_q$.

$d(q,p) = 0$: $p = q$ and $i_p^{(0)} = r_p \leq r_q$.

$d(q,p) = d+1$: There is a predecessor s of p such that $d(q,s) = d$. By induction hypothesis $i_s^{(k)} \leq r_q$ for $k \geq d$. p computes $i_p^{(d+1)}$ after having consumed $d+1$ messages $\langle i_s^{(k)} \rangle$. At least one of them has $k \geq d$, so it carries a value $\leq r_q$, and it follows $i_p^{(d+1)} \leq r_q$. □

Lemma 4.3: $i_p^{(k)} \geq I$.

Proof: Again by induction. $i_p^{(0)} = r_p \geq I$. For $k > 0$ $i_p^{(k)}$ is computed as the infimum of values, all $\geq I$ by induction, and hence $i_p^{(k)} \geq I$. □

Theorem 4.1: In each process p INF will terminate, and with the correct result $i_p = I$.

Proof: Lemmas 4.1, 4.2, and 4.3. □

Applications of protocol INF are numerous, also due to the Infimum Theory, stating that infimum operators are generic for a large class of operators:

Theorem 4.2: (Infimum Theory) Let X be a set and $\blacksquare$ be a commutative, associative, and idempotent operator on X. Then there is a partial ordering $\leq$ on X, such that $\blacksquare$ is just taking infimum regarding $\leq$.

Proof: Let X and $\blacksquare$ be given. Define $\leq$ by $x \leq y \Leftrightarrow x = x \blacksquare y$. We leave it to the reader to show that (1) $\leq$ as defined is a partial ordering (2) $\blacksquare$ is taking infimum regarding $\leq$. □

The following problems can be solved using INF:

Election: Suppose we want to select one process to perform some special, centralized action. We use the protocol INF, where X is the (totally ordered) set of possible process identities and $r_p = p$. All processes end with $i_p = m$, the smallest identity in the network. This process is the elected one.

Distributed Infimum Approximation (DIA): In [Te86] the DIA problem is defined as an abstraction of several interesting distributed problems, among which are Distributed Termination Detection [Fr80], and Global Virtual Time [Je85] (Global Cutoff [SL87]). The problem is to approximate the infimum of *changing* values x_p and [Te86] describes how this can be done,

when an algorithm for *fixed* values r_p is known. Combining the various constructions in [Te86] with different algorithms for INF (suitable for a particular class of networks) a large class of solutions for the DIA problem is described. The Distributed Termination Detection protocols by Erikson and Skyum [ES85] and Szymanski et al. [SSP85] are both based on the described INF protocol. Interesting in these protocols is that invocations of the INF algorithm overlap: in each phase a new INF invocation starts. The messages contain the phase number of the recentmost invocation whose value (so far) is *passive*. [SSP85] also provides an algorithm to compute D.

Connectivity: We want the processes to compute $\mathbb{P}$, i.e., determine what processes are in the system. Note, that set union is commutative, associative and idempotent and hence we can compute $\mathbb{P} = \text{union } \{\{p\} : p \in \mathbb{P}\}$ using INF. See next section.

Sum: Each process p has a local value r_p, and we want to compute the sum $S = \sum_{p \in \mathbb{P}} r_p$. Note + is not an idempotent operator and hence theorem 4.2 does not apply. However, we can use INF to compute the set $V = \{(p, r_p) : p \in \mathbb{P}\} = \text{union } \{\{(p, r_p)\} : p \in \mathbb{P}\}$ in each process. From V it is possible to compute S locally as the sum of the second components of V's elements. In this way it is possible to compute any function of V, like multiplicity of elements, etc. The INF protocol can thus be used to compute "complete knowledge" of the network. An interesting question is, whether a sum can be computed as elegantly and efficiently as an infimum, i.e., without using the unique process identity and with messages of small size. The PIF algorithms of [GA84], [Se83] and [Ku87] all build spanning trees in the network, directed towards one special node. Using such a tree, Sum can be done efficiently.

5 Connectivity

We want the processes to compute $\mathbb{P}$. As mentioned in the previous section, this can be done using a version of INF:

Protocol CON1:

```
Spontaneously or upon receipt of a message do:
    if phase_p = 0 then
        begin i_p := {p} ; phase_p := 1 ; shout i_p   end ;
    if all incoming queues contain a message now then
        begin consume one message from each queue ;
            i_p := union {i_p, i read in previous statement} ;
            phase_p := phase_p + 1 ;
            if phase_p ≤ D then shout i_p
                            else ready
        end.
```

Several interesting remarks can be made. If a set $\overline{\mathbb{P}}$ of potential network members is known, a set $S \subseteq \overline{\mathbb{P}}$ can be represented as a bit vector $x[\overline{\mathbb{P}}]$, where $x[p] \Leftrightarrow p \in S$. This gives us a solution to the connectivity problem, using two-valued vectors, where previous solutions ([Fi79], [Se83]) used three-valued vectors.

If a subset of $\mathbb{P}$ is represented as a list the size of the representation will be linear in the size of S. The size of a message in the CON1 protocol can be $O(N \log N)$ bits (we will always assume that an identity is $O(\log N)$ bits) and the bit complexity of CON1 is $O(D \cdot E \cdot N \cdot \log N)$. We can reduce the bit complexity to $O(E \cdot N \cdot \log N)$ if the communication is FIFO. In that case p need only shout elements of i_p that were new in this phase: old elements were shouted before and, by the FIFO property, known to the successors of p already when they consume the new elements. We modify the protocol accordingly:

Protocol CON2 (FIFO):

```
Spontaneously or upon receipt of a message do:
    if phase_p = 0 then
        begin i_p := {p} ; phase_p := 1 ; shout i_p   end ;
    if all incoming queues contain a message now then
        begin old_p := i_p ;
            consume one message from each queue ;
            i_p := union {i_p, i read in previous statement} ;
            new_p := i_p - old_p ;
            phase_p := phase_p + 1 ;
            if phase_p ≤ D then shout new_p
                            else ready
        end.
```

The identity of each process is sent exactly once over each channel, thus the bit complexity is

$O(E \cdot N \cdot \log N)$.

The $(k+1)^{th}$ shout in CON2 of p consists exactly of those processes that have distance k to p (to be proven later). It follows that if p's $(k+2)^{th}$ shout is empty (but its $(k+1)^{th}$ is not) $D_p = k$. We will use this to compute relative diameters and the network diameter later.

6 Minimum Hop Routing

A *routing table* R_p is a table specifying for each process $q \neq p$ an outgoing link $R_p[q]$ of p. p forwards any message, destined for q, over this link. The successor r of p that receives the message can either keep it (if $r = q$) or send it further via $R_r[q]$. We always want routing tables to be *loop free*, that is, a message handled this way never returns to a node where it has been before, but reaches its destination eventually. In this case the links $R_p[q]$ for all p form a directed tree routed to q, called the *in-tree* of q. The in-tree of q can be described by a table T_q, where $T_q[p] = R_p[q]$.

In this section we will see how to compute the R_p such that the routing tables satisfy the *Minimum Hop property:* the path, over which a message is sent from p to q is always a shortest possible path. This is equivalent to the T_q being *Minimum Hop in-trees:* the path from a node to the root in the tree is always a shortest possible path in the network.

We will compute the R_p in two parts. The first part, Min Hop Tree, computes the T_q in each process q. The second part, Routing Table Routing, distributes this information through the network in such a way that the R_p are constructed.

6.1 Minimum Hop Tree

We compute Min Hop Trees with a refinement of the CON2 protocol. The FIFO property of channels is essential here. Each occurrence of p in some set i_q or new_q has an outgoing link of p attached to it. This will always be the first link of a shortest path from p to q. p sends $\{(p,j)\}$ over each link j in its first shout. When q computes the union of sets containing (p,i) and (p,j), it chooses i or j nondeterministically. We obtain the following refinement of CON2:

Protocol MHT (FIFO):

```
Spontaneously or upon receipt of a message do:
    if phase_p = 0 then
        begin i_p := {p} ; T_p[p] := "deliver message to host" ;
            phase_p := 1 ; forall j ∈ Out_p do send {(p, j)} over j
        end ;
    if all incoming queues contain a message now then
        begin old_p := i_p ;
            consume one message from each queue ;
            i_p := union {i_p, i read in previous statement} ;
            new_p := i_p − old_p ;
            forall (q, j) ∈ new_p do T_p[q] := j ;
            phase_p := phase_p + 1 ;
            if phase_p ≤ D then shout new_p
                           else ready
        end.
```

Theorem 6.1: In $phase_p = d(q,p)$ p fills in $T_p[q]$ with an outgoing link of q that is the first link of a path of length $d(q,p)$ from q to p.

Proof: Again let $i_p^{(k)}$ be the set p computes directly after the kth consume. We first prove by induction on k that $q \notin i_p^{(k)}$ for $k < d(q,p)$.

$k = 0$: $i_p^{(0)} = \{p\}$ and $q \notin \{p\}$ if $0 < d(q,p)$.

$k = k'+1$: If $d(q,p) > k'+1$ then $d(q,r) > k'$ for all predecessors r of p. By induction we have $q \notin i_p^{(k')}$ and $q \notin i_r^{(k')}$. $i_p^{(k'+1)}$ is computed as the union of these sets and hence does not contain q also.

By lemma 4.2 we have $q \in i_p^{(d(q,p))}$, so q is new for p in phase $d(q,p)$ and p fills in $T_p[q]$ with the attached outgoing link of q. It remains to show that this link is the first one of a path of length $d(q,p)$ to p.

$d(q,p) = 0$: $p = q$, and p has set $T_p[p]$ to "deliver message to host" in $phase_p = 0$.

$d(q,p) = 1$: In the first phase p receives q's $\{(q,j)\}$, sent over link j. j is a path of length 1 from q to p.

$d(q,p) = d+1$: p receives (q, j_r) from one or more predecessors r with $d(q,r) = d$ and by induction j_r is the first link of a path of length d from q to r. So each of the j_r is the first link of a path of length $d+1$ from q to p.

This proves the theorem. □

So T_p is a correct Minimum Hop Tree for p. Each identity is sent once over each link (together with a link label) and thus the bit complexity of MHT is $O(E \cdot N \cdot \log N)$.

6.2 Routing Table Routing

The MHT protocol computes the information we want correctly, but not in the process where we want it. The information $R_p[q]$, needed in p, is computed in q as $T_q[p]$. We will now give two ways to distribute the information over the network so that each p constructs its complete R_p.

The most obvious way to do this is by using the PI protocol. Each q uses PI to make T_q known to all other processes. p can fill in $R_p[q]$ as soon as it receives T_q. So, N broadcasts are active concurrently and p will need an array of N *done* booleans. Instead, we will assume that $R_p[q] = undef$ initially, so we can inspect $R_p[q]$ to see whether p has shouted T_q already. q starts the protocol spontaneously as soon as it has completed the computation of T_q.

Protocol RTR1:

Upon computation or receipt of T_q:
if $R_p[q] = undef$ **then**
begin $R_p[q] := T_q[p]$; **shout** T_q **end.**

By theorem 2.1 it follows that all p will eventually fill in $R_p[q]$. Each T_q is sent over each link exactly once (Theorem 2.1) so RTR1 uses EN messages of $O(EN^2 \log N)$ bits together. Protocol RTR1 is time optimal.

For two reasons it seems likely that this can be done using fewer messages or bits. First, in RTR1 each p receives the entire T_q for each q, where it needs only one entry of it, namely $T_q[p]$. So p could do with less information. Second, remember that the T_q contain *routing information.* So we will try to use this information to do the rerouting of it more efficiently. The following protocol makes use of these ideas.

Protocol RTR2:

Step 0: Select a leader l.
Step 1: Broadcast T_l using PI.
Step 2: All q send T_q to l via a shortest path.
Step 3: l computes the R_p from the T_q.
Step 4: Each R_p is sent from l to p via a shortest path.

After running MHT, the first part of the Minimum Hop algorithm, each process knows $\mathbb{P}$. Using this information a leader can be elected easily: each p computes $l = \max(\mathbb{P})$ and plays the role of the leader iff $l = p$. So, step 0 costs no extra messages. Step 1 costs E messages of size $O(N \log N)$ bits. In step 2, each p forwards T_q as soon as it has received the broadcast of T_l. Any T_q received before T_l must be stored temporarily. Each T_q, of size $O(N \log N)$ bits, travels in at most D hops to l, so step 2 costs at most ND messages of this size. Step 3

is an internal computation in l and costs no messages. In step 4, p forwards R_q as soon as $R_p[q]$ is known, i.e., as soon as R_p is received. R_q arriving earlier than R_p must be stored. The cost of step 4 is again $\leq ND$ messages of size $O(N\log N)$ bits. In this analysis D is the actual network diameter, not just a known upper bound for it. The complexity of the steps together is $O(E+ND)$ messages of $O(N\log N)$ bits, so the bit complexity is $O((E+ND)(N\log N))$.

$O(DN^2\log N)$ is optimal here: the total amount of information is $\Theta(N^2\log N)$ bits, and each table entry can be computed at distance $\Omega(D)$ from where it is needed. Major disadvantages of RTR2 as compared to RTR1 are:

(1) It is centralized, so the system is vulnerable to failure of l. Moreover, l will have much work to do.

(2) $O(N^2\log N)$ bits of storage are needed in each node, where $O(N\log N)$ bits suffice in RTR1.

We presented the algorithm using arrays for the T and R. If $\mathbb{P}$ is not known a priori this is of course impossible. In stead, the tables could be represented as e.g. hash tables or balanced trees, containing records (q, j), where q is the search key. $R_p[q]$ is undefined if no such a record is in the structure.

7 Network diameter

In this section we describe in our framework the algorithm by Szymanski, Shy and Prywes [SSP85] to compute the network diameter. Recall $D_p = \max_q d(q,p)$. $D = \max_{p,q} d(q,p) = \max_p D_p$. The algorithm operates in two parts: first, each p computes D_p and second, the maximum of these D_p is found by using a variant of INF. Again we assume FIFO channels. Recall that in CON2 p finds $new_p = \emptyset$ for the first time in phase $D_p + 1$. So, p knows D_p as soon as it finds $new_p = \emptyset$. It then shouts $\emptyset$ to tell its successors that no further sets will follow (to avoid deadlock). Incoming messages for the first part are further ignored.

For the second part, p runs INF up to $phase2_p > D_p$. By lemmas 4.2 and 4.3 and $\forall q : d(q,p) \leq D_p$ we have $j_p = D$, the network diameter. A last shout avoids deadlock in p's successors. Again, later incoming messages for the second part are ignored. D_p is *undef* initially, $ready2_p$ signals completion of second part.

```
First part (FIFO):
    Spontaneously or upon receipt of a message i ⊆ IP do:
        if D_p ≠ undef then (* First part completed *)
            ignore message
        else begin
            if phase1_p = 0 then
                begin i_p := {p} ; phase1_p := 1 ; shout i_p   end ;
            if all incoming queues contain a i ⊆ IP now then
                begin old_p := i_p ;
                    consume one message from each queue ;
                    i_p := union {i_p, i read in previous statement} ;
                    new_p := i_p - old_p ;
                    if new_p = ∅ then D_p := phase1_p - 1 ;
                    phase1_p := phase1_p + 1 ;
                    shout new_p
                end
            end.

Second part:
    Upon computation of D_p or receipt of d ∈ IN do:
        if not ready2_p then
            begin
                wait until D_p ≠ undef ;
                if phase2_p = 0 then
                    begin j_p := D_p ; phase2_p := 1 ; shout M   end ;
                if all incoming queues contain a message d now then
                    begin consume one message from each queue ;
                        j_p := max {j_p, d read} ;
                        phase2_p := phase2_p + 1 ;
                        shout j_p ;
                        if phase2_p > D_p then ready2_p := true
                    end
            end.
```

From lemma 4.2 and 4.3 it follows that when p sets $ready2_p$ to *true* $j_p = D$. Message and bit complexity are $O(DE)$ and $O(EN\log N)$ respectively. It is an interesting and important question whether algorithms with a lower complexity for this problem exist.

8 Topological changes

All algorithms in this paper can be extended to deal with topological changes (i.e., addition or failure of nodes or links) in the same way as it is done in [Se83]. It is assumed that neighboring processes are aware of the change, In_p and Out_p sets are updated automatically and further, after any topological change that makes the network no longer strongly connected eventually a new change will follow that makes it strongly connected again.

Let A be any network protocol for strongly connected networks, that can be started by any number of processes independently. Assume A computes a value f_p in each process p, where f_p depends on the network topology G. After a topological change this value may be no longer correct and a repeated evaluation of A is necessary. The extended protocol proceeds in levels, where each level is identical to the non-extended protocol. Process p maintains its active level in L_p. p appends L_p, the level which is active in p, to every message it sends. The start of a higher level is triggered by a topological change and the execution overrides all lower levels:

Protocol EXA: (* Extended A *)
Spontaneously or upon topological change:
$L_p := L_p + 1$; start A.

Upon receipt of $<M, L>$:
if $L < L_p$ **then** discard message
else if $L = L_p$ **then** handle M according to A
else (* join A on higher level *)
begin $L_p := L$;
clear all message queues ; (* they contain lower level messages only *)
reset all state variables of A to initial value ;
handle M according to A
end.

Theorem 8.1: After a finite sequence of topological changes the system converges to a state where all L_p are equal and all $f_p = f_p(G)$, where G is the final configuration.
Proof: See [Se83]. □

All protocols except RES can be described as computing a local value depending on G, although it is not unique for Min Hop Routing. It is not hard to see that this approach works for RES also.

[Je85] Jefferson, D.R., *Virtual time*, ACM ToPLaS 7 (1985), 404-423.

[Ku87] Kutten, S., *Stepwise construction of an efficient distributed traversing algorithm for general strongly connected directed graphs*, Technion, Haifa, 1987.

[Se83] Segall, A., *Distributed network protocols*, IEEE Trans. Inf. Theory IT-29 (1983), 23-35.

[SL87] Sarin, S.K., and N.A. Lynch, *Discarding obsolete information in a replicated database system*, IEEE Trans. Soft. Eng. SE-13 (Jan. 1987), 39-47.

[SSP85] Szymanski, B., Y. Shy, and N. Prywes, *Terminating iterative solutions of simultaneous equations in distributed message passing systems*, Proc. 4th symp. on Principles of Distr. Comp., Vancouver, Canada, 1984.

[Te86] Tel, G., *Distributed infimum approximation*, Techn. Rep. RUU-CS-86-12, Dept. of Computer Science, University of Utrecht, Utrecht, 1986.

[TT87] Tel, G., and R.B. Tan, *The Equivalence of Some Network Problems*, notes July 1987.