Propagated Timestamps: A Scheme for The Stabilization of Maximum Flow Routing Protocols

Jorge A. Cobb Mohamed Waris
Department of Computer Science
University of Houston
Houston, TX 77204-3475

Abstract

We present a distributed protocol for maintaining a maximum flow spanning tree in a network, with a designated node as the root of the tree. This maximum flow spanning tree can be used to route the allocation of new virtual circuits whose destination is the designated node. As virtual circuits are allocated and removed, the available capacity of the channels in the network changes, causing the chosen spanning tree to lose its maximum flow property. Thus, the protocol periodically changes the structure of the spanning tree to preserve its maximum flow property. The protocol is self-stabilizing, and hence it tolerates transient faults. Furthermore, it has the nice property that, while the structure of the tree is being updated, no loops are introduced, and all nodes remain connected. That is, the tree always remains a spanning tree whose root is the designated node.

1. Introduction

Computer networks can be represented as connected, directed graphs, where nodes represent computers and edges represent channels between computers. Each edge is assigned a positive capacity. The flow of a network path is the minimum of the capacities of the edges in the path.

Identifying the path with the maximum flow between two nodes is useful for establishing virtual circuits in computer networks. To establish a virtual circuit with some required capacity between two nodes in a network, a maximum flow path between the two nodes is first identified. If the flow of the identified path is greater than or equal to the required capacity of the circuit, then the circuit is established along the identified path. Otherwise, the circuit is rejected.

In this paper, we describe a distributed protocol for maintaining a maximum flow spanning tree whose root is a designated node. Each new virtual circuit whose destination is the designated node is allocated along this tree. However, as virtual circuits are allocated along the tree, the tree may lose its property of being a maximum flow tree, and needs to be updated. We describe a protocol that periodically updates the structure of the spanning tree to preserve its maximum flow property. The protocol is self-stabilizing [6] [10], and has the nice property that, while the tree is being updated, it always remains a spanning tree.

Self-stabilizing protocols for maintaining a maximum flow spanning tree have also been presented in [7] and [8]. However, these protocols have the disadvantage that they either introduce temporary routing loops, or they require a termination detection algorithm. In addition, the network diameter must be known. In this paper, we present

a protocol that overcomes these disadvantages, by using a novel technique based on propagating timestamps.

We present the protocol in three steps. First, we introduce a basic protocol which is vulnerable to routing loops. Then, we refine the protocol to prevent loops from occurring during normal execution of the protocol, but is not self-stabilizing. We then present the self-stabilizing version of the protocol.

To simplify our network model, we assume each computer can read the variables of its neighboring computers. We will relax this assumption to a message passing model in the full version of the paper. Our protocols have been proven correct. Due to space limitations, proof sketches are provided in the appendix. The detailed proofs are deferred to the full paper.

2. Protocol Notation

Our network model consists of a set of processes which read but not write the variables of other processes. Two processes are neighbors iff they can read the variables of each other. The processes may be viewed as a network graph, where each node corresponds to a process, and each edge corresponds to a pair of neighboring processes. The network graph is assumed to be connected.

Each process is defined by a set of local constants, a set of local inputs, a set of local variables, and a set of actions.

Assume each process has a local variable named f. We denote with f.u the local variable f of process u. For simplicity, we omit the suffix if the process is clear from the context.

The actions of a process are separated by the symbol [], using the following syntax:

begin *action* [] *action* [] . . . [] *action* **end**

Each action is of the form *guard* → *command*. A guard is a boolean expression involving the local constants, inputs, and variables of the process, and the variables of neighboring processes. A command is constructed from sequencing (;) and conditional (**if fi**) constructs that group together **skip** and assignment statements. Similar notations for defining network protocols are discussed in [4] and [5].

An action in a process is *enabled* if its guard evaluates to **true** in the current state of the network. An *execution step* of a protocol consists of choosing any enabled action from any process, and executing the action's command. Executions are maximal, i.e., either they consist of an infinite number of execution steps, or they terminate in a state in which no action is enabled. Execution are assumed to be fair, that is, each action that remains continuously enabled is eventually executed.

If multiple actions in a process differ by a single value, we abbreviate them into a single action by introducing parameters. For example, let j be a parameter whose type is the range 0 . . 1. The action

$$x[j] < 0 \quad \rightarrow \quad y[j] := \textbf{false}$$

is a shorthand notation for the following two actions.

$$x[0] < 0 \quad \rightarrow \quad y[0] := \textbf{false}$$
$$[]\, x[1] < 0 \quad \rightarrow \quad y[1] := \textbf{false}$$

We use quantifications of the form

$$\langle \forall\ x : R(x) : T(x) \rangle$$

where R is the *range* of the quantification, and T is the *body* of the quantification. The above denotes the conjunction of all T(x) such that x is a value satisfying predicate R(x). If R(x) is omitted, then all the possible values of the type of variable x are used.

3. The Basic Protocol

We assume there exists a designated process in the network, called root. The purpose of the protocol is to build a maximum flow spanning tree (defined below) of the network graph with process root as the root of the tree.

Associated with each directed edge is a *capacity*, i.e., the available bandwidth of the edge. Edge capacities are assumed to be updated by an external agent which keeps track of the available bandwidth as virtual circuits are added to or removed from the edge.

The *flow of a directed path* is the minimum of the capacities of all the edges in the path. A directed path P beginning at process u and ending in process v is said to be a *maximum flow path* if there is no other path from u to v with a flow greater than the flow of P.

A spanning tree T is called a *maximum flow tree* iff for every process u, the path contained by T from u to the root is a maximum flow path. For routing efficiency, if for a process u there are two maximum flow paths to the root, then the protocol should include in the tree the path with the smallest number of edges.

To represent the edges in the tree, each process has a variable pr indicating its parent in the tree. Also, each process has a local constant N indicating its set of neighbors in the network graph.

To build a maximum flow spanning tree, each process maintains two variables, f and d. Variable f indicates the flow of the path to the root along the spanning tree, and variable d indicates the distance (i.e., number of edges) to the root along this path. For a pair of neighbors u and v, we define relation **better** as follows.

$$(f.u, d.u)\ \textbf{better}\ (f.v, d.v) \equiv f.u > f.v \ \vee\ (f.u = f.v \wedge d.u < d.v)$$

That is, (f.u, d.u) **better** (f.v, d.v) is true iff the path from u to the root has a greater flow than the path from v to the root, or if these flows are the same but the distance to the root from u is smaller than the distance from v. Relation **worse** is the complement of relation **better**.

To build the spanning tree, we use an algorithm similar to the Bellman-Ford algorithm, but tailored towards maximum flow paths rather than minimum cost paths. The basic procedure consists of two steps. One step is for each process u to periodically compare its (f.u, d.u) pair with that of its parent, and update (f.u, d.u) if necessary. This is because, at any time, it is possible that the flow of the path to the root via the parent changes, due to changes in the capacity of some edges along the path.

The second steps consists of choosing a new parent, if possible. For each neighbor v of u, u reads the current value of (f.v, d.v), and chooses v as its new parent iff the flow of the path to the root via v is better than the flow of the path to the root via its current parent. That is, u chooses v as its new parent iff

(**min**(f.v, cap(u,v)), d.v+1) **better** (f.u, d.u)

where cap(u,v) is the capacity of edge (u,v).

The basic protocol for a non-root process is given below. References to local variables have no suffix, while references to neighbor's variables have the neighbor's identifier as a suffix.

process u

constants

N : { w | w is a neighbor of u}

inputs

c : **array** [N] **of integer** /* c[v] = cap(u,v) */

variables

pr : **element of** N /* parent of u */
f : **integer** /* flow to the root */
d : **integer** /* distance to the root */

parameters

v : **element of** N /* v ranges over all neighbors of u */

begin

(f, d) ≠ (**min**(f.pr, c[pr]), d.pr+1) →
f, d := **min**(f.pr, c[pr]), d.pr+1

[]

(**min**(f.v, c[v]), d.v+1) **better** (f, d) →
f, d, pr := **min**(f.v, c[v]), d.v+1, v

end

The process contains two actions. In the first action, variables f and d are updated to reflect the current flow and distance to the root via the parent process pr. In the second action, a neighbor v is checked to see if it offers a path to the root with a flow greater than the flow of the current path to the root. If this is the case, v is chosen as the parent, and f and d are updated accordingly.

The root process may be specified as follows.

process root

constants

F : **integer** /* maximum flow possible in the tree */

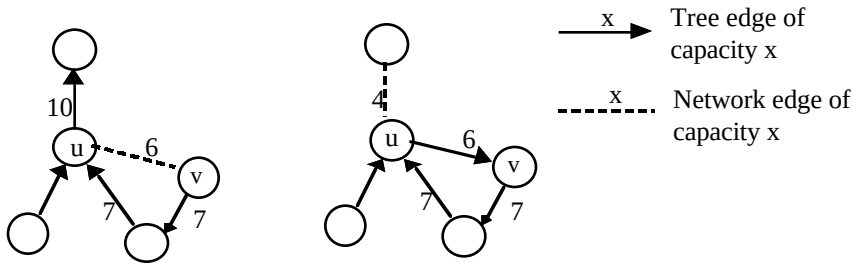
variable

pr : **process-id** /* parent of root process */
f : **integer** /* flow of the root */
d : **integer** /* distance of the root */

begin

d ≠ 0 ∨ f ≠ F ∨ pr ≠ root → d, f, pr := 0, F, root

end



a) Before edge capacity changes b) After edge capacity changes

Figure 1: Routing loop

The code contains a single action. Since the root has no parent, its parent variable should be set to itself. Also, the distance to itself should be set to zero, and the flow to itself should be set to the maximum value possible.

4. The Propagated Flow Protocol

The protocol presented in Section 3 allows processes to adapt to changes in the capacities of network edges, and to choose as a parent the neighbor that offers the best path to the root. However, there is a flaw in the protocol, which we remedy in this section.

As the capacities of edges change, a routing loop could occur, as shown in Figure 1. In the figure, the capacity of the edge between u and its parent decreases, and u chooses v as its new parent, since v appears to offer a path to the root with a flow greater than the flow of the path to the root via the current parent. However, v is actually a descendant of u, and thus a loop is formed.

There are two approaches to solve this problem. The first is to assume an upper bound D on the diameter of the network. If a process u is in a loop, then its distance d, and also the distance of all processes in the loop, will grow beyond bound. Thus, process u changes parents when d is greater than D [7], which breaks the loop. This technique has the disadvantage of requiring the protocol to have knowledge of the size of the network, and furthermore, although permanent loops do not occur, temporary loops occur while the protocol adapts the spanning tree to changes in the capacity of some edges.

The second approach, which we adopt in this paper, is to prevent loops from forming altogether. To do so, we use the technique similar to the one originally proposed in [9]. Variations of this technique have been used to prevent loops in leader election protocols [0] and routing protocols for minimum cost paths [3] [11].

We would like to ensure that for each process u and each descendant v of u in the spanning tree,

(f.u, d.u) **better** (f.v, d.v) (1)

That is, v does not have a path to the root with a better flow than the current path from u to the root. Therefore, u will not choose v as its parent, avoiding the formation of a loop.

Unfortunately, it is not always possible to ensure the above. This is because as edge capacities change, the value of (f.v, d.v) may be temporarily inaccurate, and overestimates the flow of the path from v to the root. This may cause u to incorrectly choose v as its new parent.

To resolve this, each process maintains a boolean variable, fclean, with the following meaning.

$$\text{fclean.u} \Rightarrow (\forall\ v : v \text{ is a descendant of } u :\ (f.u, d.u)\ \textbf{better}\ (f.v, d.v)) \qquad (2)$$

We say that u is *flow clean* if fclean.u is **true**, otherwise, we say that u is *flow dirty*. To ensure Relation (2) always holds, process u becomes flow dirty whenever it updates (f.u, d.u), and the new value is worse than the previous one. Furthermore, process u becomes flow clean when all its children in the spanning tree are flow clean, and the flow of u is better than the flow of each child. Therefore, if u is flow clean, it implies that the latest decrease in (f.u, d.u) has propagated to all its descendants.

Given Relation (2), process u is free to change parents if it is flow clean. If u is flow clean, and all its descendants have a flow worse than that of u, then u will not choose any of them as its new parent. Thus, loops are avoided.

We are now ready to present the definition of each non-root process u. The root process remains the same as in the previous section.

```
process u
constants
    N          :   { w | w is a n is a neighbor process of v}
inputs
    c          :   array [N] of integer /* c[v] = cap(u,v) */
variables
    pr         :   element of N   /* parent of u */
    f          :   integer        /* flow to the root */
    d          :   integer        /* distance to the root */
    fclean     :   boolean        /* flow clean bit */
parameters
    v          :   element of N   /* v ranges over all neighbors of u */
begin
    (f, d) ≠ (min(f.pr, c[pr]), d.pr+1)        →
                  if (min(f.pr, c[pr]), d.pr+1) worse (f, d)    →
                        fclean := false
                  [] (min(f.pr, c[pr]), d.pr+1) better (f, d)   →
                        skip
                  fi;
                  f, d := min(f.pr, c[pr]), d.pr+1
[]
```

fclean ∧ (**min**(f.v, c[v]), d.v+1) **better** (f, d) → f, d, pr := **min**(f.v, c[v]), d.v+1, v

[]

(∀ w : w ∈ N ∧ pr.w = u : fclean.w ∧ (f, d) **better** (f.w, d.w)) → fclean := **true**

end

The process has three actions. The first action is similar to the first action in the previous section. However, if the values of (f, d) become worse, then the process becomes flow dirty. In the second action, a new parent is chosen, but only if the process is currently flow clean. In the last action, the process becomes flow clean if all its children are flow clean and the process has a flow that is better than the flow of its children.

5. Correctness of the Propagated Flow Protocol

In this section, we characterize the behavior of the protocol of the previous section using closure and convergence properties [4]. We first define the terms closure and convergence, and then present the specific closure and convergence properties of the protocol.

An *execution sequence* of a network protocol **P** is a sequence (state.0, action.0; state.1, action.1; state.2, action.2; . . .) where each state.i is a state of **P**, each action.i is an action of some process in **P**, and state.(i+1) is obtained from state.i by executing action.i.

A *state predicate* S of a network protocol **P** is a function that yields a boolean value at each state of **P**. A state of **P** is an *S-state* iff the value of state predicate S is **true** at that state.

State predicate S is a *closure* in **P** iff at least one state of **P** is an S-state, and every execution sequence that starts in an S-state is infinite and all its states are S-states. Let S and S' be closures in **P**. We say that S *converges to* S' iff every execution sequence whose initial state is an S-state contains an S'-state.

From the above definition, if S converges to S' in **P**, and if the system is in an S-state, then eventually the execution sequence should reach an S'-state, and, because S' is a closure, the execution sequence continues to encounter only S'-states indefinitely. The converges to relation is transitive. Also, if S converges to S' and T converges to T', then S ∧ T converges to S' ∧ T'.

We next present the properties of the propagated flow protocol. We begin with a couple of definitions. Let desc(u) denote the set of *descendants* of u. That is, v ∈ desc(u) iff there is a path (w_0, w_1, w_2, ... , w_n), such that w_0 = v, w_n = u, and for every i, $0 \le i < n$, $pr.w_i = w_{(i+1)}$.

Predicate FC (Flow Clean) below relates fclean.u and the flow of u's descendants.

$$FC \equiv (\forall\ u, v : v \in desc(u) : fclean.u \Rightarrow (f.u, d.u)\ \textbf{better}\ (f.v, d.v))$$

Predicate ST (Spanning Tree) below is true iff the parent variables form a spanning tree.

$$ST \equiv (\forall\ u : : u \in desc(root)) \wedge pr.root = root$$

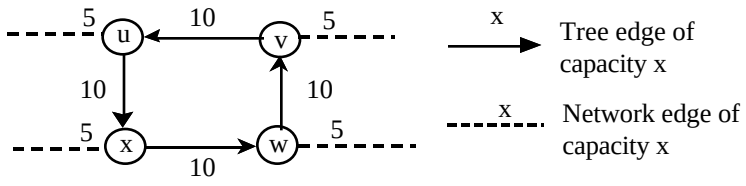


Figure 2: Routing loop

Lemma 1: FC $\wedge$ ST is a closure

♦

Thus, if the protocol begins in a state in which FC holds and the parent variables define a spanning tree, then this continues to hold forever.

Let pr_path(u) be the path obtained by following the parent pointers from u to the root. Predicate MFST below is true iff a maximum flow spanning tree is obtained, and variables f and d of each process are correct with respect to the parent.

$$MFST \equiv ST \wedge (f.root, d.root) = (F, 0) \wedge$$
$$(\forall u : u \neq root : (f.u, d.u) = (\textbf{min}(f.(pr.u), cap(u, pr.u)), d.(pr.u)+1)) \wedge$$
$$(\forall u : u \neq root : pr\_path(u) \text{ is a maximum flow path})$$

For the next theorem, we assume that edge capacities remain constant. Otherwise, the maximum flow spanning tree is a moving target that may never be reached.

Theorem 1: FC $\wedge$ ST converges to FC $\wedge$ MFST

♦

Thus, if the protocol begins in a state in which FC and ST holds, eventually a maximum flow spanning tree is found. If the edge capacities change, then the protocol will adapt and obtain a new maximum flow tree, while continuously maintaining a spanning tree.

6. The Propagated Timestamp Protocol

The propagated flow protocol presented above has a weakness. Its is very sensitive to the initial state of the system. For example, consider Figure 2. The arrows in the figure indicate the parent relationships, e.g., u has chosen x as its parent, and x has chosen w as its parent. The numbers indicate the capacity of each edge. If flow variable f of each process in the loop is currently 10, then the loop will not be broken, because all edges not included in the loop have a lower capacity than the loop edges, and all processes in the loop will not change parents.

The protocol presented in [7] is insensitive to the initial state, in the sense that it will converge to a maximum flow spanning tree irrespective of the initial state. However, it introduces temporary loops when the edge capacities change. In [8], an improved version of the protocol is presented that is insensitive to the initial state, and does not introduce temporary loops. However, it also requires knowledge of the diameter of the network, and more importantly, it requires an underlying self-stabilizing termination detection protocol.

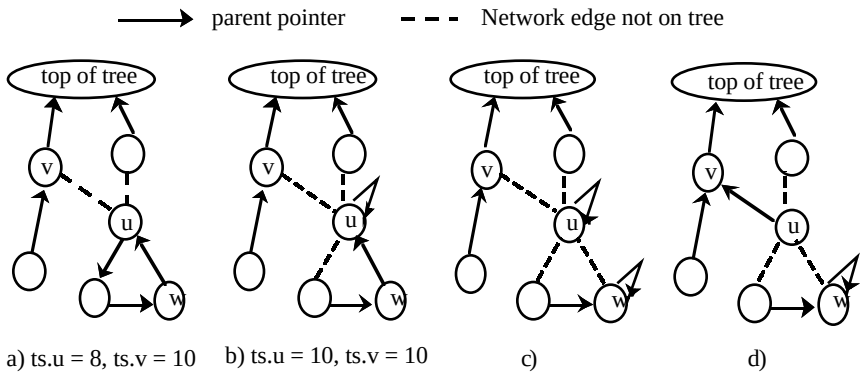


Figure 3: Breaking loops with propagated timestamps

In this section, we present a protocol that is insensitive to the initial state and does not introduce temporary loops. Furthermore, it requires no underlying termination detection protocol and no knowledge of the network diameter. It is based on propagating timestamps through the network. Using timestamps to either break loops or avoid the formation of loops in network protocols has been done in the past [1] [2]. However, we present a novel method of propagating timestamps, which allows our maximum flow protocol to be self-stabilizing.

The basic strategy is as follows. The root process has a timestamp variable, called ts.root. Periodically, the root increases its timestamp. When a child u of the root notices that the timestamp of the root, ts.root, is greater than its own timestamp, ts.u, u assigns ts.root to ts.u. Similarly, a child v of u will assign ts.u to ts.v provided ts.u > ts.v, and so on. Thus, the timestamp of the root will propagate to all its descendants.

We would like to place a bound on the difference between the timestamp of the root and that of its descendants, namely, they should differ by at most one. To accomplish this, each process u maintains a boolean variable, tclean.u, with the following meaning.

$$\text{tclean.u} \Rightarrow (\forall\ v : v \in \text{desc(u)} :\ \text{ts.v} \geq \text{ts.u}) \qquad (3)$$

We say that u is *timestamp clean* iff tclean.u equals **true**, otherwise u is *timestamp dirty*. If u is timestamp clean, it implies that all its descendants have a timestamp that is at least the timestamp of u. To guarantee Relation (3), process u makes itself timestamp dirty whenever it increases its timestamp by copying its parent's timestamp into its own. Process u makes itself timestamp clean whenever each child of u is timestamp clean and has a timestamp at least that of u.

If the root process does not increase its timestamp until it is timestamp clean, then it is guaranteed that its timestamp is always at most one greater than that of its descendants. Therefore, timestamps propagates in "waves". When all processes have a timestamp equal to k, and the root is timestamp clean, the root increases its timestamp to k+1 and becomes timestamp dirty. Then, timestamp k+1 propagates down

the tree until it reaches the leaves of the tree. The leaves then become timestamp clean (since they have no children), which in turn allow their parents to become timestamp clean, and so on, until the root becomes timestamp clean once more.

The fact that the timestamp of the root is at most one greater than all the timestamps of its descendants can be used by a process to detect that it is involved in a loop. Assume process u notices that it has a neighbor v, where ts.v ≥ ts.u + 2. This implies that u is disconnected from the root, and is most likely involved in a loop. That is, u never received timestamp ts.u+1 from its parent, and hence its parent does not lead to the root.

If the above is the case for process u, then u must choose a different parent. The steps to do so are illustrated in Figure 3. Initially (Figure 3(a)), ts.v ≥ ts.u + 2. Thus, u must change parents. Before doing so, u sets ts.u = ts.v, because the timestamp of the root is at least ts.v. Next (Figure 3(b)), u sets pr.u = u, indicating to all its neighbors that its does not have a parent. Since u no longer has a parent, each child of u, e.g. w, also sets pr.w = w (Figure 3(c)). Thus, eventually u will have no children. When this is the case, u is free to choose any parent whose timestamp is at least ts.u (Figure 3(d)) and rejoin the tree.

We are now ready to present the definition of a non-root process u.

```
process u
constants
    N        :   { w | w is a n is a neighbor process of v}     /* neighbor set */
inputs
    c        :   array [N] of integer     /* c[v] = cap(u,v) */
variables
    pr       :   element of N   /* parent of u */
    f        :   integer        /* flow to the root */
    d        :   integer        /* distance to the root */
    fclean   :   boolean        /* flow clean bit */
    ts       :   integer        /* timestamp */
    tclean   :   boolean        /* timestamp clean bit */
parameters
    v        :   element of N   /* v ranges over all neighbors of u */
begin
    (f, d) ≠ (min(f.pr, c[pr]), d.pr+1)          →
                  if (min(f.pr, c[pr]), d.pr+1) worse (f, d)     →
                        fclean := false
                  [] (min(f.pr, c[pr]), d.pr+1) better  (f, d)    →
                        skip
                  fi;
                  f, d := min(f.pr, c[pr]), d.pr+1
[]
```

tclean ∧ fclean ∧ ts = ts.v ∧ (pr.v ≠ v ∨ v = root) ∧
(**min**(f.v, c[v]), d.v+1) **better** (f, d) →
f, d, pr := **min**(f.v, c[v]), d.v+1, v

[]

(∀ w : w ∈ N ∧ pr.w = u : fclean.w ∧ (f, d) **better** (f.w, d.w)) →
fclean := **true**

[]

ts.pr > ts → tclean, ts := **false**, ts.pr

[]

(∀ w : w ∈ N ∧ pr.w = u : tclean.w ∧ ts.w ≥ ts) → tclean := **true**

[]

(ts.v ≥ ts + 2 ∨ pr.pr = pr) ∧ pr ≠ u →
pr, ts, tclean := u, **max**(ts, ts.v), **false**

[]

(∀ w : w ∈ N ∧ pr.w ≠ u) ∧ pr = u ∧ (pr.v ≠ v ∨ v = root) ∧ ts.v ≥ ts →
fclean, tclean, ts := **true**, **true**, ts.v;
pr, f, d := v, **min**(f.v, c[v]), d.v+1

end

Process u contains seven actions. As before, the first action updates the pair (f, d) to match those of its parent, and becomes flow dirty if (f, d) became worse. The second action changes parents. We have strengthened the guard to ensure that u is timestamp clean, that the new parent has a timestamp equal to u's, and that the new parent also has a parent of its own. The third action is the same as in the propagated flow protocol.

The fourth action updates the timestamp to that of the parent, and makes u timestamp dirty. The fifth action makes u timestamp clean. The sixth action detects that a neighbor v has ts.v ≥ ts + 2, or that the parent of u has no parent. In this case, u could be in a loop, so it sets pr to u. The timestamp of u is increased to at least the timestamp of the neighbor.

In the last action, u rejoins the tree. If u has no parent and no children, and it finds a neighbor who does have a parent and whose timestamp is at least u's timestamp, then u chooses that neighbor as its new parent, it updates its timestamp and flow to match those of its new parent, and becomes flow and timestamp clean.

The specification root process is similar to that of previous sections, except that we require one additional action, which is given below.

(∀ w : w ∈ N ∧ pr.w = root : tclean.w ∧ ts.w ≥ ts) → ts := ts+1

In this action, the root increases its timestamp if all its children are timestamp clean and have a timestamp greater than or equal to the root's timestamp.

7. Correctness of the Propagated Timestamp Protocol

We next show that the propagated timestamp protocol is self-stabilizing, that is, regardless of the initial state of the system, the protocol will converge to a maximum

flow spanning tree. Furthermore, during any fault-free execution of the protocol, the integrity of the spanning tree is maintained, even while the tree is adapting to changes in edge capacities.

We show the correctness in several steps. We first present some closure properties of the protocol, and then prove that it converges to a maximum flow spanning tree. We begin by noting that the flow clean predicate is restored automatically regardless of the initial state of the system.

Lemma 2: a) FC is a closure
b) **true** converges to FC

♦

We next consider the timestamp clean bit. Predicate TC (Timestamp Clean) below relates tclean.u and the timestamps of the descendants of u.

$$TC \equiv (\forall\ u, v : v \in\ desc(u) : tclean.u \Rightarrow ts.v \geq ts.u\ )$$

Lemma 3: a) TC is a closure
b) **true** converges to TC

♦

Since both FC and TC are closures, and they both are restored automatically, then TC $\wedge$ FC is also restored automatically (**true** converges to TC $\wedge$ FC). However, this does not ensure that loops are broken, since FC and TC can hold even in the presence of loops. To show that loops are broken, we need the following theorem.

Theorem 2: TC $\wedge$ FC converges to TC $\wedge$ FC $\wedge$ ST

♦

Thus, by transitivity of the converges relation, **true** converges to FC $\wedge$ TC $\wedge$ ST, i.e., the spanning tree is restored automatically.

For the next theorem, we assume that edge capacities remain constant. Otherwise, the maximum flow spanning tree is a moving target that may never be attained.

Theorem 3: TC $\wedge$ FC $\wedge$ ST converges to TC $\wedge$ FC $\wedge$ MFST

♦

By transitivity, **true** converges to TC $\wedge$ FC $\wedge$ MFST, and the protocol is self-stabilizing.

8. Further Refinements

Several further refinements are possible to the propagated timestamp protocol. Most important of these are to relax the model to a message passing system, and to obtain a version of the protocol that uses only a timestamp with a finite number of values rather than an unbounded timestamp.

It can be shown that the convergence time of the protocol is $O(D^2)$, where D is the diameter of the network. Convergece time may be reduced to O(D) with the following simple refinement. Each process maintains an additional variable, maxts, which is set to the maximum of both the ts and maxts variables of itself and of its neighbors. This will cause the value of maxts of the root to quickly obtain the value of the maximum timestamp in the system. The root would then set its timestamp to

a value at least maxts, which quickly ensures that no node has a timestamp greater than the root.

The above and other practical refinements will be presented in the full version of the paper.

References

[0] A. Arora and A. Singhai, "Fault-Tolerant Reconfiguration of Trees and Rings in Networks", *IEEE International Conference on Network Protocols*, 1994, page 221.

[1] A. Arora, M. G. Gouda, and T. Herman, "Composite Routing Protocols", Proc. of the Second IEEE Symposium on Parallel and Distributed Processing, 1990.

[2] J. Cobb, M. Gouda, "The Request-Reply Family of Group Routing Protocols", to appear in *ACM Transactions on Computers*, 1998.

[3] J. J. Garcia-Luna-Aceves, "Loop-Free Routing Using Difussing Computations", *IEEE/ACM Transactions on Networking*, Vol 1, No. 1., Feb. 1993, page 130

[4] M. Gouda, "Protocol Verification Made Simple", *Computer Networks and ISDN Systems*, Vol. 25, 1993, pp. 969-980.

[5] M. Gouda, *The Elements of Network Protocols*, textbook in preparation.

[6] M. Gouda, "The Triumph and Tribulation of System Stabilization", *International Workshop on Distributed Algorithms*, 1995.

[7] M.Gouda and M. Schneider, "Stabilization of Maximum Flow Trees", Invited Talk, *Proceedings of the third Annual Joint Conference on Information Sciences*, 1994, pp. 178-181. A full version was submitted to the journal of Information Sciences.

[8] M. Gouda and M. Schneider, "Maximum Flow Routing", *Second Workshop on Self-Stabilizing Systems*, 1996.

[9] P. M. Merlin and A. Segall, "A Failsafe Distributed Routing Protocol", *IEEE Transactions on Communications*, Vol. COM-27, No. 9, pp 1280-1288, 1979.

[10] M. Schneider, "Self-Stabilization", ACM Computing Surveys, Vol. 25, No. 1, March 1993.

[11] A. Segall, "Distributed Network Protocols", *IEEE Transactions on Information Theory*, Vol. IT-29, No. 1, pp. 23-35, Jan. 1983.

Appendix

A.1 Properties of the Propagated Flow Protocol

Proof Sketch of Lemma 1:

One can show that FC by itself is a closure. The proof is very similar to the proof of part a) of Lemma 2 in the propagated timestamp protocol. To show that FC $\wedge$ ST is a closure, note that the only time ST can be invalidated is when a process u changes parents. Since the new parent has a flow better than that of u, and u is flow clean, then, from FC, the new parent cannot be a descendant of u. Thus, a loop is not formed and ST is maintained.

♦

For the next theorem, we assume that edge capacities remain constant.

Proof Sketch of Theorem 1:

Since FC $\wedge$ ST is a closure, the parent variables define a spanning tree, and continue to do so forever. We have to show that eventually this spanning tree is a maximum flow spanning tree.

Define T to be the following tree of processes. Initially, T contains only the root. Let u be the child of the root whose edge capacity to the root is the highest. Let this capacity be C.

We first show that all processes eventually have a flow at most C. We begin with the children of the root. If a child updates its flow to that of the edge to the root, then its flow becomes at most C. Furthermore, any child that the root gains will have a flow equal to the edge to the root, and hence at most C. Therefore, all children of the root will eventually have a flow at most C, and this will continue to hold forever.

We must then prove that if all descendants of the root down to level L of the tree have a flow at most C, then this continues to hold forever. This can be shown by executing each action and checking if the new state satisfies the above. We omit the details.

We then must show that if all descendants of the root down to level L of the tree have a flow at most C, then so will processes at level L+1, and this will continue to hold forever. Again, we omit the details.

By induction, all processes eventually have a flow at most C. Since u is a neighbor (distance 1) from the root, it will choose the root as its parent. Let T now be the root plus process u.

We repeat the same argument again, but finding the neighbor v of T whose edge into T has the greatest capacity C' into set T, and argue that all nodes outside of T eventually have a flow at most C', and that v chooses its neighbor in T as its parent.

By construction, a maximum flow tree is the result.

♦

A.2 Properties of the Propagated Timestamp Protocol

Proof Sketch of Lemma 2:

Part a)

Let prlen(v,u), where v is a descendant of u, be the number of edges in the path from v to u following the parent variables. We define FC(i) as follows:

$$\text{FC(i):}\ \ (\forall\ u, v : v \in \text{desc(u)} \wedge \text{prlen(v,u)} \leq i : \text{tclean.u} \Rightarrow \text{(f.u, d.u) } \textbf{better} \text{ (f.v, d.v)})$$

We must show that FC(i) is a closure for all i, and hence FC is also a closure. To show that FC(i) is a closure, execute each action in the protocol under the assumption that FC(i) holds before the action and show that FC(i) holds after the execution.

Part b)

We show by induction that FC(i) eventually holds for all i regardless of the initial state.

Base case: i = 1. Consider any process v. Process v will eventually execute its action to update its flow to that of its parent u. Then, its flow is worse than that of its parent, and hence FC(1) holds between v and u. It its easy to show that FC(1) continues to hold forever between v and its parent, even if it changes parents.

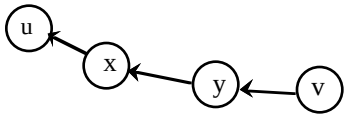


Figure 4: Restoring FC(i+1) between u and v.

Induction case: assume FC(i) holds, show that eventually FC(i+1) holds.

Consider any process v, as illustrated in Figure 4, and assume i = 2. Since we assume FC(i) holds, the relationship between u and each of x and y satisfies FC(i). Descendant v will eventually execute its action to update its flow to that of its parent. When this occurs, if u is flow clean, then the flow of y is worse than that of u, and hence the new flow of v is worse than u's, and FC(i+1) holds between u and v.

Assume next that v was not a descendant of u, and either v or an ancestor (say y) chose to join the subtree of u. If this is the case, since the process changing parents (y in this case) must be flow clean, from FC(i), v's flow is worse than y, which in turn is worse than x's. Thus, FC(i+1) holds between u and v.

Hence, for any descendant v of u at a distance i+1 from u, either FC(i+1) holds between v and u when v joins the subtree of u, or v is already in the subtree and FC(i+1) holds between v and u eventually.

We need to show next that FC(i+1) continues to hold between v and u. This can be shown in a manner similar to that of part a) by going through all the actions in the protocol assuming FC(i+1) holds between v and u and showing it continues to hold after executing the action.

Therefore, FC(i+1) holds eventually between u and any descendant v at a depth of i+1, which implies that eventually FC(i+1) holds between any pair of processes and continues to hold forever.

♦

Lemma 3: The proof is very similar to that of Lemma 2 and is omitted.

Proof Sketch of Theorem 2:

We prove the theorem in two parts. For the first part, we show that the timestamp of the root always increases. For the second part, we prove that eventually a spanning tree is formed.

Part one.

To show that the timestamp of the root always increases, we must show that the guard of the action that increases it has to become true. The guard checks two things. First, all children must have a timestamp at least that of the root. Second, that all children are timestamp clean. The first part will become true since because of fairness, all children must eventually execute the action that sets their timestamp to that of their parent.

To prove that all children eventually become timestamp clean, we show the following. Define, for all i,

clean_ts(i) = ($\forall$ u : u $\in$ desc(root) $\wedge$ prlen(u, root) $\leq$ i :
all processes in pr_path(u) from the root down to the first timestamp clean process have non-decreasing timestamps)

It can be shown by induction that each clean_ts(i) will hold and continues to hold until the root increases its timestamp. Thus, timestamps will be non-decreasing from the root up to the first timestamp clean process in every rooted path. A simple induction argument shows that all processes will become timestamp clean, and thus the root is free to increase its timestamp.

Part two:

Knowing that the timestamp of the root always increases, we need to show that a spanning tree is obtained. From Lemmas 2 and 3, we know that eventually FC and TC hold and continue to hold. We assume that FC and TC already hold. Let the largest timestamp of any process be T at this time.

By part one, eventually the root increases its timestamp to T+1, and no other process has a timestamp greater than T. What we would like to show is that eventually all processes have a timestamp greater than T+1, and that at all times all processes with timestamps greater than T form a tree.

When the root increases its timestamp to T+1, we have a tree with a single node. Since when a process changes parents the new parent must have the same timestamp as the process, and thus a process with timestamp at least T+1 will never choose as a parent a process with timestamp less than T+1. Furthermore, since FC and TC hold, no process ever chooses a descendant as its parent. Hence, all processes with timestamp at least T+1 always form a tree.

What remains to be shown is that all processes eventually have a timestamp of at least T+1.

From TC, when the root is timestamp clean, all processes have a timestamp at least ts.root. However, it is easy to argue that the root timestamp is always at least that of any other process. Thus, if a process has a timestamp at least T+1, then its timestamp is either equal to the root (when the root is timestamp clean) or one less than the root (when the root is timestamp dirty).

Since the timestamp of the root always increases, eventually it becomes greater or equal to T+3. In which case, for each process, either the process has a timestamp at least T+2 (and thus is connected to the root) or has a timestamp at most T (and thus not connected to the root).

Any process v with timestamp T or less with a neighbor whose timestamp is T+2 or greater will execute action six, setting pr.v = v. Eventually, all its children do the same, and v ends with no parent and no children. Thus, v is now free to execute action seven and join the tree, and set its timestamp to at least T+2 (i.e., the timestamp of its new parent). Since all processes eventually have a timestamp of at least T+1 (in effect, at least T+2), and all these processes form a tree, and continue to form a tree, the theorem holds.

♦

Theorem 3: The proof is very similar to that of Theorem 1 and is omitted.