MAXIMUM FLOW ROUTING

Mohamed G. Gouda and Marco Schneider

Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712-1188

Abstract

We describe a protocol for routing and allocating virtual circuits from any vertex to a designated destination in a computer network. In this algorithm, the network vertices maintain a maximum flow spanning tree whose root is the designated destination. Every requested virtual circuit is allocated along the maintained tree. However, as virtual circuits are allocated along the tree, the tree may lose its property of being a maximum flow tree, and so need to be updated. We describe a novel protocol for the network vertices to periodically update the maintained tree to keep it a maximum flow tree. This protocol is stabilizing, and has the nice property that while the tree is being updated, it always remains a tree whose root is the designated destination.

1. Introduction

Computer networks can be represented as connected, undirected graphs, where vertices represent computers and edges with positive capacities represent channels between computers. The flow of a path in such a network is the minimum capacity for an edge in that path.

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

Also, if an edge in an established circuit happens to fail, a maximum flow path between the two ends of the failed edge is identified. Then the two disjointed parts of the circuit are re-connected by a connection along the identified path [4].

In this paper, we describe a protocol for allocating virtual circuits from any vertex to a designated destination in a computer network. In our protocol, the network vertices maintain a maximum flow spanning tree whose root is the designated destination. Every requested virtual circuit is allocated

along the maintained tree. However, as virtual circuits are allocated along the tree, the tree may lose its property of being a maximum flow tree, and need to be updated. We describe a protocol for periodically updating the maintained tree to keep it a maximum flow tree. This protocol is stabilizing [8], and has the nice property that while the tree is being updated, it always remains a tree.

2. Maximum Flow Trees

A network N is a connected and undirected graph whose set of vertices is V and whose set of edges is E. One vertex r in V is called the root of N. Associated with each edge $\{v, w\}$ in E is a non-negative integer $c.\{v, w\}$, called the capacity of edge $\{v, w\}$.

A path in N is a non-empty sequence of distinct vertices in V such that each pair of consecutive vertices in the sequence is an edge in E. A path is called rooted iff its last vertex is the root r of N. For example, the path that consists of the single vertex r is rooted.

The flow of a rooted path in N is the minimum capacity of an edge in the path. Thus, if a rooted path p in N is $\langle v_0 ; \ldots ; v_k \rangle$, then

the flow of p =

$$\min \{ c.\{v_i, v_{i+1}\} \mid 0 \leq i < k \}$$

A rooted path p in N is called a maximum flow path iff for every rooted path q that has the same first vertex as p in N,

the flow of p $\geq$ the flow of q

The flow of a vertex v, other than the root r, in N is the flow of a maximum flow path, whose initial vertex is v, in N. The flow of r is the maximum edge capacity in N.

A spanning tree T of N is called a maximum flow tree iff every rooted path in T is a maximum flow path in N.

The following theorem, taken from [7], establishes the relation between the maximum weight spanning trees of a network [5] and the maximum flow trees of the same network.

Theorem 1: Each maximum weight spanning tree of a network N is a maximum flow tree of N. The converse is not necessarily true.

A corollary of this theorem is that each network has a maximum flow tree.

3. Maximum Flow Tree Protocol

In this section, we present a distributed protocol, taken from [7], for maintaining a maximum flow tree T in an arbitrary network N = (V, E). The protocol is distributed because it consists of several programs, one for each vertex in V, and the program of each vertex v has few constants that relate to vertex v only. In particular, the program of a vertex v has the following four constants:

i. D is an upper bound on the number of vertices in the longest path in N.

ii. F is the maximum edge capacity in network N.

iii. H is the set H of neighbors of vertex v in network N.

iv. c[w] is the capacity for each edge {v, w} incident at vertex v in N.

The program of each vertex v also has three variables: p.v, d.v, and f.v. When the protocol terminates, p.v is the parent of vertex v in the maintained maximum flow tree T, d.v is the number of edges on the path from vertex v to the root r in tree T, and f.v is the flow of v.

The program of each vertex v has two actions as follows.

begin <action> [] <action> **end**

Each of the two actions is of the following form.

<guard.w> --> <statement.w>

where w is a parameter that denotes a neighbor of v, <guard.w> is a boolean expression over the variables of v and the variables of w, and <statement.w> is a sequence of assignment statements that assign values to variables of v based on the current values of the variables of v and the variables w.

An action "<guard.w> --> <statement.w>" is enabled (for execution) when there is a neighbor w of v that makes the boolean expression <guard.w> true. An enabled action is executed by executing the sequence of assignment statements in its <statement.w>.

The program of any vertex v, other than the root r, in network N is as follows.

constant

D : **integer**,
F : **integer**,
H : **set** {w|w is a neighbor of v in N },
c : **array** [H] **of** 0..F /* c[w] = capacity of {v, w} */

variable

p.v : H, /* parent of v in T */
d.v : 0..D, /* distance of v from r in T*/
f.v : 0..F /* flow of v in T*/

parameter

w : H /* a neighbor of v*/

begin

p.v = w ^
(d.v ≠ min {d.w + 1, D} or f.v ≠ min {f.w, c[w]})
-->
d.v := min {d.w + 1, D};
f.v := min {f.w, c[w]}

[] p.v ≠ w ^
d.w < D-1 ^ (d.v = D or (d.w + 1 - D*min {f.w, c[w]}) < (d.v - D*f.v))
-->
p.v := w;
d.v := min {d.w + 1, D};
f.v := min {f.w, c[w]}

end

This program has two actions. In the first action, vertex v detects that its distance d.v from the root or its flow f.v is not consistent with that of its parent. In this case, vertex v updates both d.v and f.v to be consistent, respectively, with d.w and f.w of its parent w.

In the second action, vertex v detects that it has a neighbor w such that the following three conditions hold. First, w is not the current parent of v. Second, the distance from the root to w is less than D-1. Third, if w is to become the parent of v, then v will either have more flow or will have the same flow and a smaller distance from the root. In this case, vertex v makes w its parent and updates its variables d.v and f.v accordingly.

The program for the root vertex r has only two constants, D and F, and two variables, d.r and f.r. This program is as follows.

constant

D : **integer**,

F : **integer**

variable

d.r : 0..D, /* distance of r from r */

f.r : 0..F /* flow of r in T */

begin

$d.r \neq 0$ or $f.r \neq F$ --> $d.r := 0$; $f.r := F$

end

The one action in this program ensures that variables d.r and f.r are assigned the two values 0 and F, respectively.

A proof of the correctness and stabilization of this protocol is presented in [7]. This proof is similar to those given in [1], [2], and [3].

4. Routing Using the Maximum Flow Tree Protocol

From the above protocol, each vertex v in network N knows its parent vertex p.v in the maintained maximum flow tree T. When a virtual circuit is to be established through a vertex v (to the root vertex r), vertex v routes the circuit through its parent vertex p.v in the maximum flow tree. In other words, the circuit is established over the edge {v, p.v}.

Each established virtual circuit is of some capacity. Thus, establishing a virtual circuit with a capacity cp decreases the remaining capacity in each edge along the circuit by the value cp. Similarly, when an established circuit with capacity cp is removed, the remaining capacity in each edge along the circuit is increased by the value cp.

Each vertex v keeps track of the remaining capacity cr[w] for each edge {v, w} incident at v in the network. Vertex v follows the next two rules to keep its remaining capacity array cr current.

i. When a virtual circuit with capacity cp is established over an edge {v, w}, then vertex v updates its cr array as follows.

cr[w] := cr[w] - cp

Also, vertex w updates its cr array as follows.

cr[v] := cr[v] - cp

(This can happen only when both cr[w] in v and cr[v] in w are greater than cp.)

ii. When a virtual circuit with capacity cp that was established over an edge {v, w} is removed, then vertex v updates its cr array as follows.

cr[w] := cr[w] + cp

Also, vertex w updates its cr array as follows.

cr[v] := cr[v] + cp

As the remaining capacity cr arrays become very different from the capacity c arrays used in maintaining the maximum flow tree, the maintained tree is no longer a maximum flow tree. Therefore, the maintained tree needs to be updated periodically.

One protocol for updating the maintained maximum flow tree is as follows. Each node in the network periodically uses its cr array to update its c array by executing the statement c := cr. Unfortunately, whenever a node executes such a statement, the original maximum flow tree protocol may start executing and cause the maintained tree to lose its property of being a tree for some time, until the protocol finally converges to a new tree.

In this paper, we are interested in protocols for updating the maintained maximum flow tree such that the maintained tree is always a tree, even when it is being updated. One such protocol is discussed in the next section.

5. Adaptive Maximum Flow Tree Protocol

In this section, we present a protocol for adapting the maximum flow tree to changes in the remaining channel capacities. This protocol proceeds in successive rounds: round.0, round.1, round.2, ...

In each odd round (i. e. round.1, round.3, ...), every vertex uses its cr array to update its c array, and enables its first action in the original maximum flow tree protocol. Note that these first actions, when executed, do not change the maintained tree. They merely compute the correct flow along every path in the maintained tree. The round terminates when the values of variables d.r and f.r are 0 and F, respectively, and the values of variables d.v and f.v of each vertex v, other than the root r, are consistent with those of its parent w in the maintained tree as follows.

d.v = min{d.w + 1, D}

f.v = min{f.v, c[w]}

Recall that at the beginning of an odd round, each c array is assigned the value of the corresponding cr array. Thus at the end of the round, the maintained tree may not be a maximum flow tree with respect to the new c arrays.

In each even round (i. e. round.0, round.2, ...), every vertex enables its two actions in the original maximum flow tree protocol. Although the maintained tree may be updated in this round, it can be shown that each action execution in this round keeps the maintained structure a tree. The round terminates when the values of variables p.v, d.v, and f.v of every vertex v define a maximum flow tree.

To keep track of the current round, each vertex v has a variable s.v whose value is the number of rounds executed so far. To determine whether the current round is odd or even, vertex v needs only to check whether or not s.v **mod** 2 equals 0.

Before initiating the next round, the root vertex r waits long enough until all activities in the current round have ceased. Then the root r increments its variable s.r to start the next round. Each other vertex v starts to participate in the next round, when it observes that the value of s.w of a neighbor w is larger than the value of its own s.v.

The program of any vertex v, other than the root r, in network N is as follows.

constant

D	:	**integer**,	
F	:	**integer**,	
H	:	**set** {w|w is a neighbor of v in N }	

input

cr	:	**array** [H] **of** 0..F	

variable

c	:	**array** [H] **of** 0..F,	
p.v	:	H,	/* parent of v in T */
d.v	:	0..D,	/* distance of v from r in T*/
f.v	:	0..F,	/* flow of v in T*/
s.v	:	**integer**	/*seq. number of round*/

parameter

w : H /*a neighbor of v*/

begin

/*start an odd round*/

s.v < s.w ^ s.w **mod** 2 ≠ 0

-->

s.v := s.w;

c := cr

[] /*execute in each round*/

s.v = s.w ^

p.v = w ^

(d.v ≠ min {d.w + 1, D} or f.v ≠ min {f.w, c[w]})

-->

d.v := min {d.w + 1, D};

f.v := min {f.w, c[w]}

[] /*start an even round*/

s.v < s.w ^ s.w **mod** 2 = 0

-->

s.v := s.w

[] /*execute in each even round*/

s.v = s.w ^ s.v **mod** 2 = 0 ^

p.v ≠ w ^

d.w < D-1 ^ (d.v = D or (d.w + 1 - D*min {f.w, c[w]}) < (d.v - D*f.v))

-->

p.v := w;

d.v := min {d.w + 1, D};

f.v := min {f.w, c[w]}

end

The program of vertex v has four actions. In the first and third actions, vertex v detects that the value of variable s.w of a neighbor w is larger than its own s.v, indicating that the next round has started. In this case, v assigns its s.v the value of s.w, and if s.w **mod** 2 ≠ 0 indicating that the new round is odd, v uses its cr array to update its c array.

The second action in this program is the same as the first action in the original maximum flow tree protocol except for adding the conjunct (s.v = s.w) to the guard. The fourth action in this program is the same as the second action in the original maximum flow tree protocol except for adding the conjunct (s.v = s.w ^ s.v **mod** 2 = 0) to the guard.

The program of the root vertex r is as follows.

constant

D : **integer**,
F : **integer**,
H : **set** {w|w is a neighbor of r in N }

variable

d.r : 0..D, /* distance of r from r */
f.r : 0..F /* flow of r in T*/
s.r : **integer**

parameter

w : H /*a neighbor of r*/

begin

timeout no other action in the network is enabled --> s.r := s.r + 1

[] s.r < s.w --> s.r := s.w

[] d.r ≠ 0 or f.r ≠ F --> d.r := 0 ; f.r := F

end

The program of the root r has three actions. In the first action, when r is certain that the activities of the current round have terminated, it starts a new round by incrementing the value of s.r by one. This action is called a timeout action because it can be implemented using time-outs as discussed in Section 7. In the second action, r observes that its s.r has a smaller value than s.w of some neighbor w. This observation indicates that an error has occurred, as the value of s.r should always be equal or larger than the value of s.v for any vertex v in the network. Thus, r corrects the observed error by assigning s.v the value of s.w. The third action of r is identical to the third action of r in the original maximum flow tree protocol.

6. Correctness of the Adaptive Tree Protocol

In this section, we discuss some interesting properties of the adaptive maximum flow tree protocol. Our presentation starts with some definitions concerning the states, transitions, and computations of this protocol.

A state of the adaptive maximum flow tree protocol is defined by a value for each variable in each vertex in the protocol. The value of a variable is from the domain of values for that variable.

A state of the protocol is called odd, or even, iff in that state, variable s.r in the root r has an odd value, or even value, respectively.

A state of the protocol is called balanced iff in that state, the following two conditions are satisfied.

i. For every vertex v, s.v = s.r, where r is the root.

ii. Each action of each vertex, other than the first action of the root, is disabled (and so cannot be executed).

It is straightforward to show that the following two conditions are satisfied in any balanced state.

i. For the root r,

$$d.r = 0$$
$$f.r = F$$

ii. For every vertex v, other than the root r, there is a neighbor w such that

$$s.v = s.r$$
$$p.v = w$$
$$d.v = \min \{d.w + 1, D\}$$
$$f.v = \min \{f.w, c[w]\}$$

A balanced state is called a milestone iff in that state, the values of the p.v variables, where v is any vertex other than the root r, define a tree whose root is r.

A milestone state is called a target iff in that state, the values of the p.v variables, where v is any vertex other than the root r, define a maximum flow tree whose root is r.

A triple (s, c, s') is called a transition of the protocol iff the following three conditions are satisfied.

i. s and s' are two states of the protocol.

ii. c is an action of some vertex in the protocol.

iii. Executing action c when the protocol is in state s yields the protocol in state s'.

States s and s' are called the pre-state and post-state of the transition, respectively.

An infinite sequence (s.1, c.1, s.2, c.2, ...) is called a computation of the protocol iff the following three conditions are satisfied.

i. s.1, s.2, ... are states of the protocol.

ii. c.1, c.2, ... are actions in some processes in the protocol.

iii. Every triple (s.i, c.i, s.(i+1)) in the sequence is a transition of the protocol.

Next, we present five interesting properties of an arbitrary computation of the protocol. Consider an arbitrary computation C of the protocol, and let for i = 0, 1, ... , T.i denote the i-th occurrence of a transition, if any, whose action is the timeout action of the root r, in computation C. Also, let S.i be the pre-state of transition T.i, for i = 0, 1, ...

Lemma 1: For every k, k = 0, 1, ..., computation C has a transition T.k.

Lemma 2: S.0 is a balanced state.

Lemma 3: If S.0 is an odd balanced state, then for every k, k = 0, 1, ... ,

S.(2*k + 1) is an even target state, and

S.(2*k + 2) is an odd milestone state.

Lemma 4: If S.0 is an even balanced state, then S.1 is an odd balanced state, and for every k, k = 0, 1, ... ,

S.(2*k + 2) is an even target state, and

S.(2*k + 3) is an odd milestone state.

Lemma 5: In each state, that occurs after the first target state in computation C, the values of the p.v variables define a tree whose root is r.

7. Refinements of the Adaptive Tree Protocol

The adaptive maximum flow tree protocol in Section 5 admits a number of refinements that can make it attractive in some of applications. In this section, we briefly discuss three of these refinements.

First, this protocol can be extended to allow a network to have a multiple maximum flow trees such that each vertex in the network is the root of some tree. With this extension, virtual circuits can be

established from any vertex to any other vertex in the network. This extension can be achieved by making variables p.v, d.v, f.v, and s.v for every vertex v arrays, rather than single variables, as follows.

variables

p.v : **array** [V] **of** H,
d.v : **array** [V] **of** 0..D,
f.v : **array** [V] **of** 0..F,
s.v : **array** [V] **of integer**

In these declarations, V is the set of all vertices in the network. Thus, p.v[w] is the parent of vertex v in the tree whose root is w, d.v[w] is the distance from vertex v to vertex w in the tree whose root is w, and so on.

Second, there are two methods to implement the timeout action of the root r in the adaptive maximum flow tree protocol. In one method, time-outs are used as follows. After the root r increments its variable s.r, it waits long enough until it is certain that all activities in the new round have terminated, before it times out and increment s.r one more time to initiate the next round. In the other method, the protocol is augmented with an algorithm for detecting that all actions, other than the timeout action of the root, have terminated. Several stabilizing termination detection algorithms have been proposed earlier, for example [6] and [10], and any of them can be augmented with our protocol to implement the timeout action.

Third, instead of using an integer sequence number s.v for every vertex v, binary sequence numbers can be used in the adaptive maximum flow tree protocol. However, to keep the protocol stabilizing, the protocol needs to be augmented with another protocol for maintaining a shortest path tree whose root is r. (One such protocol is discussed in [2].) Let q.v be the parent of any vertex v, other than the root r, in the shortest path tree. Then, the conjunct (s.v < s.w) in the guards of the first and third actions of vertex v is replaced by the guard (s.v ≠ s.w ^ q.v = w). It is straightforward to show that the resulting protocol is stabilizing.

References

[1] A. Arora and M. G. Gouda, "Distributed Reset", IEEE Transactions on Computers, Vol. 43, No. 9, September 1994, pp 1026-1039.

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

[3] N. S. Chen, F. P. Yu, and S.T. Huang, "A Self-Stabilizing Algorithm for Constructing Spanning Trees", Information Processing Letters, Vol. 39, pp. 147 - 151, 1991.

[4] C. E. Chow, J. D. Bickell, and S. Syed, "Performance Analysis of Fast Distributed Link Restoration Algorithms", Accepted in the International Journal of Digital and Analog Communications Systems, 1994.

[5] T. H. Cormen, C. E. Leiserson, and R. L. Rivest, Introduction to Algorithms, MIT Press and McGraw-Hill, 1990.

[6] M. G. Gouda and M. Evangelist, "Convergence/Response Tradeoffs in Concurrent Systems", Proc. of the Second IEEE Symposium on Parallel and Distributed Processing, 1990.

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

[9] M. Schneider, Ph. D. Dissertation, The University of Texas at Austin, in preparation, 1995.

[10] G. Varghese, "Self-Stabilization by Counter Flushing", Proceedings of the 1994 ACM Symposium on Principles of Distributed Computing.