Balanced Routing

Jorge A. Cobb
Department of Computer Science
University of Houston
Houston, TX 77204-3475
cobb@cs.uh.edu

Mohamed G. Gouda
Department of Computer Sciences
The University of Texas at Austin
Austin, TX 78712-1188
gouda@cs.utexas.edu

Abstract

The distance vector routing protocol satisfies the following property. If this protocol is used to route a sequence of data messages from a source to a destination, then these data messages will follow the same shortest-distance path from the source to the destination. In this paper, we show how to modify this protocol, without adding new messages, in order to satisfy the following load balancing property. If the modified protocol is used to route a sequence of data messages from a source to a destination, then these messages will be uniformly distributed over all k-monotonic paths from the source to the destination, i.e., over all paths whose distance to the destination never increases at each hop, and may remain constant in at most k hops. In particular, by choosing k to be zero, the modified protocol distributes the data messages over all shortest-distance paths from the source to the destination.

1. Introduction

Routing protocols are fundamental for computer networks. Their objective is to find an efficient path from a source to a destination, and to route data messages along this path. Due to its impact on network performance, the routing protocol must be chosen carefully. The chosen protocol must be fault-tolerant, due to the possible changes in the topology of the network, and it must respond quickly to such changes. It must be efficient, both in terms of message overhead and space overhead, in order to avoid interfering with the resources necessary to store and forward data messages.

There are mainly two types of routing protocols: [9] [11] distance-vector routing, used in the RIP Internet protocol [7], and link-state routing, used in the OSPF Internet protocol [8].

In link-state routing, each computer determines which computers are its neighbors, and whether or not the channels between itself and its neighbors are operational. Each computer then periodically broadcasts this local topology information to all other computers in the network. Since each computer receives the local topology information of all other computers, it is able to build a graph of the whole network topology, and compute the most efficient path from itself to every other computer in the network.

In distance-vector routing, each computer maintains a distance vector with its current estimate of the distance (or cost) to each destination. Each computer periodically sends a copy of this vector to each of its neighbors. By comparing its distance vector with that of its neighbors, the computer determines which neighbor is the closest to each destination.

Both link-state routing and distance-vector routing may suffer performance degradation if all data messages are always routed via the same shortest path to the destination. That is, the shortest path becomes congested. Thus, performance is significantly improved if data messages are routed via multiple paths to the destination. Two techniques to do so are emergency-exits in link-state routing [12], and multiple disjoint paths to the destination in distance-vector routing [10]. However, these techniques either require the network to be flooded with additional messages (as in [12]) or increases the complexity and overhead of the routing algorithm (as in [10]).

In this paper, we propose a simple and efficient technique to perform load balancing in a computer network whose routing protocol is based on distance-vector routing. The technique requires very small changes to the distance-vector routing protocol. It requires no additional message overhead, and requires only a small amount of additional storage overhead, namely, a small bit-map (two bits per neighbor) for every destination in the routing table.

The technique consists of uniformly distributing the data messages along each path to the destination that satisfies the following constraint. In each hop along the path, the distance to the destination either decreases or remains the same. Furthermore, the number of hops where the distance remains the same is bounded by a constant, thus ensuring that each message reaches its destination.

We organize the paper as follows. In Section 2, we present the protocol notation used throughout the paper. The notation consists of a shared memory model in which computers may read but not write the variables of its neighbors. We chose this shared memory model because it simplifies significantly the exposition of the protocols.

0-8186-8061-X/97 $10.00

We relax this model into a message passing model in a later section.

In Sections 3 and 4, we present two protocols based on the distance-vector approach. The purpose of these protocols is to collect routing information and determine the neighbors which offer the best path to the destination. The first protocol is the well-known distance vector routing protocol, and the second is our improved protocol which maintains additional information for purposes of load-balancing.

In Section 5, we describe in detail how the routing information collected by the protocol of Section 4 may be used to route data messages to the destination in a balanced manner. In Section 6, we present the protocol in detail using a message passing model. In Section 7, we consider how explicit congestion notification between neighbors may be used as an aid in load balancing. In Section 8, we present concluding remarks and future work.

2. Protocol Notation

The network consists of a set of processes, which read but do not write the variables of other processes. Two processes are neighbors iff they read the variables of each other. Let u and v be neighbors. Expression z.v in the code of process u yields the current value of variable z of process v.

The network may be viewed as a graph, where each process is a node in the graph, and each pair of neighboring processes correspond to an edge in the graph. The network graph is assumed to be connected.

Each process is defined by a set of global constants, a set of local inputs, a set of local variables, and a set of actions. 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 constants, inputs, and variables of the process. A command is constructed from sequencing (;) conditional (**if fi**), and looping (**for rof**) constructs that group together **skip** and assignment statements. Similar notations for defining network protocols are discussed in [5] and [6].

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, i.e., 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 → y[j] := **false**

is a shorthand notation for the following two actions.

x[0] < 0 → y[0] := **false**
[] x[1] < 0 → y[1] := **false**

3. Single Path Protocol

In this section, we present the well-known distance-vector routing protocol, which has been studied and proven correct in the literature [1] [2] [3] [4]. This protocol finds a single path from each source process to each destination process, and is the foundation of the RIP routing protocol being used in the Internet [7]. To simplify the exposition, we present a version of the protocol that finds a path to a specific destination process. Extending the protocol to find a path to each destination process is straightforward and is presented in Section 6.

We assume there exists a designated process in the network, namely, process r. The objective of the protocol is to build a *shortest-distance spanning tree* (defined below) of the network graph, with process r as the root of the tree.

A path is *rooted* iff r is the last process in the path. A path P from process u to process v is a *shortest-distance path* iff there is no path from u to v whose number of edges is less than those of P. A spanning tree T is a *shortest-distance spanning tree* iff every rooted path in T is a shortest-distance path.

To represent the tree edges, each process maintains in variable par the identity of the neighbor it has chosen as its parent in the tree. An edge (u, v) is a *parent edge* iff u has chosen v as its parent in the tree, i.e., par.u = v.

In addition to its parent variable, each process maintains in variable dis its current estimate of the distance to the root. Each process computes this distance by simply adding one to its parent's distance.

To choose a neighbor as its parent, each process follows a simple greedy strategy: the neighbor with the smallest distance is chosen.

The single path protocol for a non-root process u can be defined as follows.

process u
constants
  D : **integer** {upper bound on network diameter}
inputs
  N : **set** { v | v is a neighbor of u }
variables
  dis : 0 . . D {distance to the root}
  par : **integer** {parent of u}
parameters
  v : N {v ranges over all neighbors}
begin
  dis ≠ min(dis.par + 1, D) → dis := min(dis.par +1, D)
[] dis > dis.v + 1 → par := v;
  dis := dis.v + 1
end

The above process contains two actions. In the first action, the process compares its distance estimate with the distance estimate of its parent. If these disagree, u's distance is corrected. This step is necessary because, at any time, the distance to the root via the parent may change due to changes in the topology of the network.

In the second action, the process checks if a neighbor is closer to the root, i.e., has a smaller distance, than the current parent. If this is the case, the process chooses this neighbor as its new parent, and updates its distance accordingly.

The root process r can be defined as follows.

process r
constants
  D : **integer** {upper bound on network diameter}
variables
  dis : 0 . . D {distance to the root}
begin
  dis ≠ 0 → dis := 0
end

Since process r is the root, it has no parent, and thus, no variable par. However, it does require a distance variable dis, whose value should always be zero.

The single path protocol satisfies the following theorem, which has been proven in [1] and [2].

Theorem 1
Regardless of the initial state of the variables of each process, within O(D) execution steps, the protocol reaches a stable state in which:
a) the parent edges define a shortest-distance spanning tree.
b) d.r = 0.
c) for each parent edge (u, v), dis.u = dis.v + 1.

♦

Theorem 1 shows the correctness and fault-tolerant properties of the protocol. If the network topology changes due to faulty channels, the protocol will update its parent and distance variables to reflect a new shortest-distance spanning tree.

4. Multiple Paths Protocol

We next extend the protocol of the previous section to discover all shortest-distance paths to the destination, rather than discovering just a single shortest-distance path. In addition, it will discover paths where the distance to the root either decreases or remains constant with each hop.

Let *distance.u* be the length of a shortest-distance path from u to the root r. Hence, dis.u is the current estimate of distance.u maintained by process u.

Note that for any network edge (u, v),

$$0 \leq |\ \text{distance.u} - \text{distance.v}\ | \leq 1.$$

Note also that if for edge (u, v), distance.u > distance.v, then (u, v) is an edge in some shortest-distance path from u to the root.

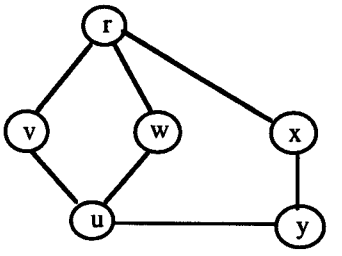

Figure 1

Thus, the neighbors of a process u can be divided into three disjoint sets: the *parents*, whose distance is smaller than u's distance, the *siblings*, whose distance is equal to u's distance, and the *children*, whose distance is greater than u's distance.

The set of parents of u consists of those neighbors which are the parent of u in some shortest-distance spanning tree. The set of siblings of u consists of those neighbors which, although they may not have the same parent as u, have the same depth in each shortest-distance spanning tree as the depth of u.

Each process u maintains two variables of type set, namely, prn and sbn. These correspond to the set of parents and the set of siblings of u, respectively.

Consider the network in Figure 1. There are two shortest-distance paths from process u to the root process r, namely, (u, v, r) and (u, w, r), and one shortest-distance path from y to the root, namely, (y, x, r). Hence, prn.u = {v, w} and prn.y = {x}. In addition, sbn.u = {y} and sbn.y = {u}, since the distances of u and y are equal.

Sets prn and sbn may be used in the routing of data messages towards the root as follows. For load balancing, each process will distribute the data messages to each of its parent neighbors in a uniform manner. However, to enhance load balancing, each process will on occasions forward data messages to its sibling neighbors. Although in doing so the data messages do not decrease their distance to the root, their distance does not increase either. This load balancing approach will be described in more detail in Section 5.

The multiple path protocol can be defined as follows.

process u
constants
  D : **integer** {upper bound on network diameter}
inputs
  N : **set** { v | v is a neighbor of u },
variables
  dis : 0 . . D {distance to the root}
  par : **integer** {parent}
  prn : **subset of** N {parent neighbors}
  sbn : **subset of** N {sibling neighbors}
parameters
  v : N {v ranges over all neighbors}

begin

dis ≠ min(dis.par + 1, D) → dis := min(dis.par + 1, D)

[] dis > dis.v + 1 → par := v; dis := dis.v + 1

[] v ∉ prn ∧ dis = dis.v + 1 → prn := prn ∪ {v}

[] v ∈ prn ∧ dis ≠ dis.v + 1 → prn := prn - {v}

[] v ∉ sbn ∧ dis = dis.v → sbn := sbn ∪ {v}

[] v ∈ sbn ∧ dis ≠ dis.v → sbn := sbn - {v}

end

The process has four actions. The first two actions are the same as in the single path protocol. Recall that, from Theorem 1, variable dis will contain the shortest-distance to the root. Thus, in the next two actions, neighbors are added to, or removed from, set prn depending on whether their distance plus one is equal to, or not equal to, the distance of u, respectively. Similarly, the last two actions add or remove neighbors from set sbn depending on whether their distance is equal to or not equal to the distance of u, respectively.

The root process remains the same as in the single path protocol.

Notice that since the values of variables par and dis are not affected by variables prn and sbn, then Theorem 1 still holds for this protocol. Thus, the neighbors in prn are the next-hop neighbors of every shortest-distance path to the root, and sbn are the neighbors whose shortest-distance to the root equals that of the process.

Notice also that in the above protocol, the neighbor par will always be a member of set prn. Furthermore, any member of prn could have been the parent of the process, since the distance to the root is the same for all members. This implies that variable par is not necessary. That is, each element of prn can be viewed as a parent of the process. We thus remove variable par, and substitute the first two actions of the above protocol by the single action below.

prn = ∅ ∨ dis > dis.v + 1 →
dis := min(dis.v + 1, D);
prn := {v};
sbn := ∅

In this action, if there is a neighbor whose distance is less than the distance of the elements of prn, then this neighbor is chosen as the new parent. That is, the sole element of prn is this neighbor, and the distance is updated to reflect the new parent's distance. If other neighbors have the same distance as the new parent, they will later be added to set prn by the third action. Since the distance has changed, set sbn is emptied. If any neighbors are siblings, they will later be added to set sbn by the fifth action.

This final version of the protocol satisfies a property similar to Theorem 1 for the single path protocol, which we state below. The proof of the theorem is given in the appendix.

Theorem 2

Regardless of the initial state of the variables of each process, within O(D) execution steps, the protocol reaches a stable state in which:

a) for each process u, distance.u = dis.u.
b) for every network edge (u, v), v ∈ prn.u iff distance.u = distance.v + 1.
c) for each network edge (u, v), v ∈ sbn.u iff distance.u = distance.v.

♦

From Theorem 2, if a data message is forwarded to a neighbor in set prn, then the message is forwarded along a shortest-distance path to the root. Hence, the distance to the root will decrease for the message. If the message is forwarded to a neighbor in set sbn, then the distance to the root remains constant for the message.

Notice that the additional space required is small. Two bits per neighbor suffice to implement sets prn and sbn.

5. Achieving Balanced Routing

In this section, we describe in more detail how data messages are routed towards the root. Each process will create data messages and receive data messages from its neighbors. The process should forward these data messages to its neighbors so that the number of edges traversed by each data message is as small as possible, while at the same time attempting to distribute these messages uniformly throughout the network to avoid congestion. The specification of this protocol using a message passing model will be given in the next section.

Consider Figure 2. Process u has two parent neighbors, namely, v and w. For load balancing, u should distribute the data messages evenly between neighbors v and w. Process u may accomplish this by selecting at random between u and w whenever it forwards a data message.

If we restrict data messages to be forwarded only to the parent neighbors, then this severely restricts the routing choices. For example, y only has a single parent neighbor x, and thus it has no routing freedom. Furthermore, the traffic of both z and u is routed through v, which may cause congestion. Process y could take advantage of its sibling neighbor u and route some of its traffic to u, and u could also take advantage of y and route some of its traffic to y.

We thus conclude that when a process receives its data message, it should choose a process at random from the union of its parent neighbors and its sibling neighbors, and forward the message to the chosen neighbor.

The above, however, has a weakness. Although unlikely, it is possible that a message is perpetually routed to sibling neighbors, and never becomes closer to the root. For example, in Figure 2, a data message could be

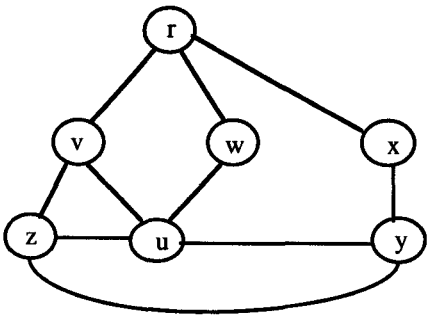

Figure 2

forwarded along the path (u, y, z, u, y, z, . . .) and never reach the root.

To resolve this problem, each data message contains a count field which is decreased by one every time the message is routed to a sibling neighbor. Once this count reaches zero, the message is forbidden from being forwarded to sibling neighbors. From then onwards, the message will be forwarded only to parent neighbors, which decreases the distance to the root. In this way, if a data message is created at process u, where distance.u = d, and the count of the message is k, then the data message will arrive to the root within d + k hops.

We say that a rooted path P is a *k-monotonic* path iff, for all edges (u, v) in P, distance.u ≥ distance.v, and furthermore, the number of edges (u, v) in P where distance.u = distance.v is at most k.

It follows from this definition that the route taken by a data message whose count is initially k will be a k-monotonic path. If the count of a message is zero, then the message is routed via a shortest-distance path to the root. If the count is k, where k > 0, then the message will be routed via any k-monotonic path to the root.

The value chosen for this count affects both delay and load balancing. If the count is zero, the delay is minimized, because the message is routed via shortest-distance path to the root. However, this could cause congestion in processes which have only a single shortest-distance path to the root. On the other hand, if the count is non-zero, a larger number of routing paths exist, which will alleviate congestion, but at the expense of increasing the delay by a few hops.

Corollary 1

Regardless of the initial state of the variables of each process, within a finite number of execution steps, the protocol reaches a stable state in which:

a) for every k-monotonic path P, and for every edge (u, v) in P, if distance.u > distance.v, then v ∈ prn.u.

b) for every k-monotonic path P, and for every edge (u, v) in P, if distance.u = distance.v, then v ∈ sbn.u.

♦

Corollary 1 implies that after the multiple path protocol reaches a stable state, then a process will route its data

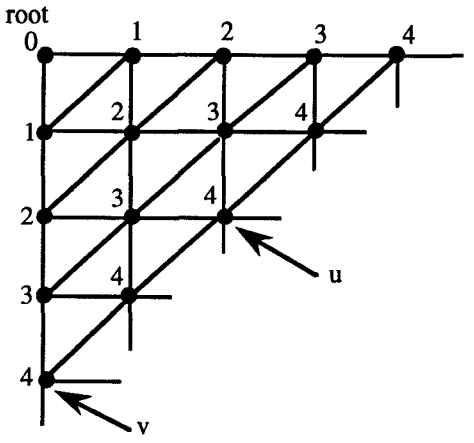

Figure 3

messages with a count equal to k uniformly over every k-monotonic path originating at the process.

We next give a more concrete example of the advantages of this form of routing. Consider Figure 3, in which processes are arranged as a mesh, and the number in each process corresponds to the distance to the root. In addition to the usual mesh edges, the network contains some diagonal edges.

Consider process u in the third row and in the third column, whose distance to the root is four. It has a total of six shortest-distance paths to the root. Thus, if the count of each data message is initially set to zero, the data messages originating at this process will be uniformly distributed over all six paths.

Assume now that the count of each data message is eight. In this case, there are a total of 48 8-monotonic paths from this node to the root, and the data messages will be uniformly distributed over these 48 paths.

More drastic is the case of process v in the fifth row and the first column, whose distance to the root is also four. It has only a single minimum distance path to the root, but it has a total of 48 8-monotonic paths to the root.

6. Balanced Routing Protocol

In this section, we present the full version of the balanced routing protocol. It consists of the multiple paths protocol of Section 4, with the additional feature of creating and routing data messages as described in the previous section. In addition, we allow every process in the network to be the destination of a data message.

The protocol is presented using a message passing model. To this end, we enhance the notation as follows. For every pair of neighboring processes u and v, there is a FIFO channel from u to v and a FIFO channel from v to u. The statement **send** msg(var) **to** v in process u appends a message of type msg to the channel from u to v, and the field in the message is the current value of variable var in process u.

In addition to boolean expressions, guards in each process u are allowed to be of the form **rcv** msg(var) **from any** v. This guard is enabled iff there is a message of type msg at the head of an incoming channel of u. If the command of an action with this receive guard is chosen for execution, then, before the command is executed, the message is removed from the channel, and its field is copied into the local variable var. Furthermore, variable v is set to the identity of the neighbor from which the message is received.

Each process has an input set Z with the identities of all the processes in the network. Note that since each process is a possible destination, there is no longer a single root process. Thus, each process z, $z \in Z$, is the root of the shortest-distance spanning tree over which data messages are routed if their destination is z.

Since now each process is allowed to be a destination, variables prn, sbn, and dis, become arrays, whose indices are the elements of Z, and the value of each element is a subset of the neighbor set N. The inputs and variables of each process are now defined as follows.

```
inputs
  N    : set { v | v is a neighbor of u },
  Z    : set { v | v is a process in the network }
variables
  prn  : array [ Z ] of subset of N,
         {parent neighbors to each destination}
  sbn  : array [ Z ] of subset of N,
         {sibling neighbors to each destination}
  dis  : array [ Z ] of 0 . . D,
         {distance to each destination}
```

Thus, if $z \in Z$, prn[z] stores the parent neighbors to reach destination z, sbn[z] stores the sibling neighbors to reach destination z, and dis[z] stores the distance to reach destination z.

The balanced routing protocol can now be defined as follows.

```
process u
constants
  D    : integer   {diameter of the network}
  cmax : integer   {max. number of sibling hops}
inputs
  N    : set { v | v is a neighbor of u },
  Z    : set { v | v is a process in the network }
variables
  prn  : array [ Z ] of subset of N,
         {parent neighbors to each destination}
  sbn  : array [ Z ] of subset of N,
         {sibling neighbors to each destination}
  dis  : array [ Z ] of 0 . . D,
         {distance to each destination}
  d    : array [ Z ] of 0 . . D, {neighbor distance}
  v    : N,                      {neighbor}
  z    : Z,                      {message destination}
  c    : 0 . . cmax              {message sibling hops}
begin
   true          →    z := any;
                      c := any;
                      RTMSG
[] rcv data(z, c) from any v →
                      RTMSG
[] true          →    for each v do
                          send upd(dis) to v
                      rof
[] rcv upd(d) from any v →
                      UPDTBL
end
```

In the first action the process creates a data message. It first chooses a destination and an upper bound on the number of sibling hops, and then routes the created data message using statement RTMSG, which is defined below.

In the second action, the process receives a data message and then routes the message to a neighbor using statement RTMSG.

In the third action, the process sends a copy of its distance array to each of its neighbors.

In the last action, the process receives a copy of the distance array from one of its neighbors, and updates its parent, sibling, and distance tables using statement UPDTBL, defined below.

Statement RTMSG is defined as follows

```
if z = u   →   skip   {arrived, deliver message}
[] z ≠ u ∧ (prn[z] = ∅ ∨ dis[z] = D)   →   skip
                          {destination not reachable}
[] z ≠ u ∧ prn[z] ≠ ∅ ∧ dis[z] < D   →
     {destination is reachable}
     if c = 0  →  v := random(prn[z])
     [] c > 0  →  v := random(prn[z] ∪ sbn[z]);
                  if v ∈ sbn[z]   →   c := c - 1
                  [] v ∉ sbn[z]   →   skip
                  fi
     fi;
     send data(z, c) to v
fi
```

In this statement, the process checks first if the destination is itself, and also if the message is not deliverable. If the message is deliverable, it checks the count in the message. If the count is zero, it chooses a neighbor at random from the parent neighbors of the destination. If the count is non-zero, it chooses a neighbor at random from the union of the parent neighbors and the sibling neighbors of the destination.

Statement UPDTBL is defined as follows.

```
dis[u] := 0;
for each z, where z ≠ u, do
  if d[z] + 1 < dis[z] ∨ prn[z] = ∅          →
                    dis[z] := min(d[z] + 1, D);
                    prn[z] := {v};
                    sbn[z] := ∅
  []¬(d[z] + 1 < dis[z] ∨ prn[z] = ∅)        → skip
  fi;
  if v ∉ prn[z] ∧ dis[z] = d[z] + 1          →
                    prn[z] := prn[z] U {v}
  [] v ∈ prn[z] ∧ dis[z] ≠ d[z] + 1          →
                    prn[z] := prn[z] - {v}
  [] (v ∈ prn[z] ∧ dis[z] = d[z] + 1) ∨
    (v ∉ prn[z] ∧ dis[z] ≠ d[z] + 1)         → skip
  fi;
  if v ∉ sbn[z] ∧ dis[z] = d[z]              →
                    sbn[z] := sbn[z] U {v}
  [] v ∈ sbn[z] ∧ dis[z] ≠ d[z]              →
                    sbn[z] := sbn[z] - {v}
  [] (v ∈ sbn[z] ∧ dis[z] = d[z]) ∨
    (v ∉ sbn[z] ∧ dis[z] ≠ d[z])             → skip
  fi;
rof
```

In this statement, the process checks for three cases for every destination process z in the network. In the first case, if the neighbor v from which the distance vector d was received has a distance smaller than the distance of the parent neighbors, then v is chosen as the single parent neighbor, and the distance is updated.

In the second case, the process checks if neighbor v should be added to or removed from the parent neighbors set according to its current distance to the destination z. In the last case, the process checks if neighbor v should be added to or removed from the sibling neighbors set according to its distance to the destination.

7. Balanced Routing with Congestion

In the previous section, the process routes the message to any parent or sibling neighbor when the count in the message is greater than zero. In this section, we assume neighboring processes inform each other about their congestion status, such as their average buffer utilization. Thus, each process learns whether each of its neighbors is congested or not. If a neighbor is congested, the process attempts to send data messages to non-congested neighbors.

To accomplish this, we introduce a new input, cgn, which indicates which neighbors of the process are experiencing congestion. The new input and a new auxiliary variable are defined as follows.

```
inputs
  cgn    : subset of N  {neighbors with congestion}
variables
  i      : subset of N
```

The statement RTMSG can now be defined to take advantage of congestion information as follows.

```
if z = u   → skip    {arrived, deliver message}
[] z ≠ u ∧ (prn[z] = ∅ ∨ dis[z] = D)   → skip
                     {destination not reachable}
[] z ≠ u ∧ prn[z] ≠ ∅ ∧ dis[z] < D     →
                     {destination is reachable}
    if c = 0 ∧ (prn[z] - cgn) = ∅   → i := prn[z]
    [] c = 0 ∧ (prn[z] - cgn) ≠ ∅   →
                     i := prn[z] - cgn
    [] c > 0 ∧ ((prn[z] U sbn[z]) - cgn) = ∅   →
                     i := prn[z] U sbn[z]
    [] c > 0 ∧ ((prn[z] U sbn[z]) - cgn) ≠ ∅   →
                     i := (prn[z] U sbn[z]) - cgn
    fi;
    v := random(i);
    if v ∈ sbn[z]   → c := c - 1
    [] v ∉ sbn[z]   → skip
    fi;
    send data(z, c) to v
fi
```

If the data message is deliverable, then the process routes the message as before, according to the count c of the message, with the exception that the message is not routed to congested neighbors. For example, if the count is non-zero, then it randomly routes the message to a parent or sibling neighbor that is not congested. If all parent and sibling neighbors are congested, then one of them is chosen randomly.

8. Concluding Remarks

In this paper, we have presented a new routing strategy based on the well-known distance vector routing. The strategy does not require any additional overhead in terms of message passing, and requires very little overhead in terms of storage. Namely, the storage required is a small bit-map for each entry in the routing table, and the bit-map consists of two bits per neighbor.

There are many directions possible for future work. One is to perform simulations with different topologies, different source locations, and different traffic generation distributions, to determine the appropriate value for the count field in each data message to improve load balancing and minimize delay.

Another direction of future work is to generalize the algorithm to allow edges to have different costs, and find all the minimum cost paths to the destination. Currently, each edge is considered to have a cost of one, and thus the shortest path to the destination is also a minimum cost path. However, in this case, the definition of k-monotonic needs to be generalized. Rather than considering the siblings to be all neighbors whose cost to the destination is equal to the process' own cost, the siblings could be all

neighbors whose cost is within a small constant from the cost of the process. This is necessary since it is unlikely that those neighbors will have a cost precisely equal to the cost of the process.

References

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

[2] Arora A., Gouda, M., "Distributed Reset", *Lecture Notes in Computer Science 472: Foundations of Software Technology and Theoretical Computer Science,* pp. 316-331, Springer Verlag, 1990.

[3] Bellman R. E., *Dynamic Programming,* Princeton, NJ: Princeton University Press, 1957.

[4] Ford L. R. Jr., Fulkerson D. R., *Flows in Networks,* Princeton, NJ: Princeton University Press, 1962.

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

[6] Gouda M., *The Elements of Network Protocols*, textbook in preparation, 1998.

[7] Hedrick C., "Routing Information Protocol," *InternetRequest for Comments* 1058, June 1988. Available from http://www.ietf.cnri.reston.va.us.

[8] Moy J., "Ospf Version 2", *Internet Request for Comments* 1583, March 1994. Available from http://www.ietf.cnri.reston.va.us.

[9] Peterson L., Davie B., *Computer Networks: A systems Approach,* 1st Edition, Morgan Kaufmann publishers, 1996.

[10] Sidhu D., Nair R., Abdallah S., "Finding Disjoint Paths in Networks", *Proceedings of the 1991 ACM SIGCOMM Conference.*

[11] Tanenbaum Andrew, *Computer Networks,* 3rd Edition, Prentice Hall publishers, 1996.

[12] Wang Z., Crowcroft J., "Shortest Path First with Emergency Exits", *Proceedings of the 1990 ACM SIGCOMM Conference.*

Appendix

Proof of Theorem 2

Let T.i be the set of processes v such that distance.v ≤ i. We show by induction over i that eventually parts (a) and (b) of Theorem 2 hold for processes in T.i, and continue to hold forever. Also, all processes z not in T.i have dis.z ≥ i, and this holds forever. Once parts (a) and (b) hold for all i, it follows immediately that part (c) will hold and continue to hold forever.

For the base case, i = 0, and T.0 = {r}. After r executes its single action, dis.r = 0. Also, all processes z other than r have dis.z ≥ 0. This continues to hold forever.

Next, assume parts (a) and (b) of Theorem 2 hold for i, each process z not in T.i has dis.z ≥ i, and this holds forever. We show that this will also be the case for i+1.

Let z be any process not in T.i. Note that all neighbors w of z will always have dis.w ≥ i. If dis.z = i and prn.z ≠ ∅, no neighbor is added to prn.z, and all elements are removed eventually by the fourth action, which is continuously enabled as long as prn.z ≠ ∅ and dis.z = i. Thus, eventually prn.z = ∅. Then, the first action is enabled, and continues to be enabled since no neighbor can be added to prn.z. Thus, eventually the action is executed, setting dis.z > i and prn.z ≠ ∅. Note that dis.z > i holds forever, since all neighbors w have dis.w ≥ i.

Let distance.u = i+1, i.e., u has a neighbor v in T.i with dis.v = distance.v = i. From above, eventually dis.u > i. Assume first that dis.u ≥ i+2. The first action of u is enabled, because dis.v = i. It remains enabled since dis.v doesn't change, and if u chooses a neighbor z not in T.i as a new parent, then dis.u ≥ i+2 remains, since dis.z > i, (shown above). Thus, the action executes eventually, setting prn.u = {v}, dis.u = i+1. We consider this next.

Let dis.u = i+1, and an element v of T.i is in prn.u. Notice that u never changes dis.u, i.e., dis.v = i, so v cannot be removed from prn.u, and each neighbor w has dis.w ≥ i. This disables the first action. Any neighbor z not in T.i (thus dis.z > i) will be removed from prn.u by the third action, and all neighbors of u in T.i will be added to prn.u in the second action. Also, neighbors not in T.i are never added to prn.u. Hence dis.u = i+1 remains true forever, and prn.u obtains and maintains its correct value.

Assume next that dis.u = i+1, and no element of T.i is in prn.u. Then one of two things will occur. If an element of T.i is added to the set, then we have the above case. Otherwise, prn.u eventually becomes empty. That is, each element z of prn.u has dis.z > i = dis.u-1, and is removed from the set. Similarly, a neighbor not in T.i cannot be added to prn.u. Then, the first action is executed, yielding two choices. If the new parent is in T.i, then dis.u = i+1 and we have the case considered earlier. Otherwise, dis.u ≥ i+2, which was also considered above.

Thus, Theorem 2 parts (a) and (b) hold for T.(i+1) and continue to hold, and all processes not in T.(i+1) have dis ≥ i+1. By induction, Theorem 2 holds for all T.i.