Link-reversal algorithms, cont'd

Readings:
Gafni, Bertsekas: Distributed algorithms for generating loop-free routes in networks with frequently changing topology.
Busch, Surapaneni, Tirthapura: Analysis of link reversal algorithms for mobile ad hoc networks.
Park, Corson TORA paper.
Rao et al. Geographical routing without location info.
Fonseca et al. Beacon vector routing.
Fang et al. GLIDER: Gradient Landmark-based distributed routing.

Next time:
Fang et al. GLIDER: Gradient Landmark-based distributed routing.
Geographic routing:
Ko, Vaidya LAR paper
Ko, Vaidya multicast paper (skim only)
Kranakis, Singh, Urrutia
Bose et al. Routing with guaranteed delivery
Karp, Kung GPSR paper

1 Introduction

Last time I presented many of the Gafni-Bertsekas and Busch results on link-reversal routing algorithms. I'll finish that up now, and then we will go on to location-free routing.

2 Gafni-Bertsekas and Busch, cont'd

Recall the problem from last time: We are given an undirected graph network, with a distinguished "destination" node d. The network may change over time. At any point in time, each edge is oriented one way or the other. We assume that they start out forming a DAG, and the algorithms all preserve this property. The goal is, given an arbitrary DAG, to gravitate to a destination-oriented one, in which all paths lead to d.

GB have three algorithms: Full reversal (both abstract, in terms of links, and concrete, in terms of heights of the form (α, u)), partial reversal, and a more general algorithm. Last time we discussed the full reversal algorithm, in both forms, and the partial reversal algorithm in the abstract form only.

I described this slightly incorrectly last time, so I will start by giving the corrected version. Then we'll finish with the concrete partial reversal algorithm and the general algorithm. I'll just touch on TORA briefly.

2.1 Abstract partial reversal algorithm

Here we'll present the GB partial reversal algorithm.

Each node $u \neq d$ maintains a list of its neighbors v that have reversed the connecting edge (u, v) since the last time u fired. (Last time, I specified neighbors that have fired—but that isn't the same as saying that they reversed the edge, since not every edge need be reversed.) Initially, all these lists are empty. Then at each step, some set of sinks fire.

For each such sink u:
If there exists at least one neighbor of u that isn't on u's current list, then u reverses the directions of exactly the links (v, u) for which v isn't on the list and empties the list. If there is no such neighbor, that is, u's list contains all of N_u, then u reverses all its incident edges. In either case, every node v such that (v, u) gets reversed adds u to its list.

We have the same five claims as before:
Claim 1. If two nodes u and v both reverse in the same step, then u and v aren't neighbors.
Proof as before.

Claim 2. At each stage, the graph is acyclic.
Proof: Now we can't use the same argument as before, because we might not be reversing all the edges, and hence might create a cycle. To prove this solely in terms of the abstract formulation, we need some new invariant. (E.g., for any node u, none of its listed neighbors is reachable by a directed path from any of its unlisted neighbors. Can this be proved by induction?
It remains to be done; it's not in the paper.)

Claim 3. If the graph is connected, then the algorithm eventually terminates, ending up with a DAG that is destination-oriented.
Proof: We did a proof, which proceeded by showing that if it doesn't terminate, then all the nodes must fire infinitely many times. That's a contradiction because d never fires. QED Claim 3

Exercise: Can you get a time bound from this, by introducing a suitable metric?

The proof of Lemma 1 in BG is actually for their general algorithm. It requires a special hypothesis (A.3, a sort of liveness property). We'll talk about this more below. BST do prove termination here, by proving time and work bounds; however, these are stated and proved in terms of the concrete version, below. There are some funny issues about initialization, as you will see.

Claim 4: If a node starts out having a directed path to d, then it will never reverse the direction of its links.
Proof as before.

Claim 5: Any two executions of the abstract partial reversal algorithm in a connected network, starting from the same global state (which includes not just link directions but also list information) are "equivalent": Each node performs the same number of reversals in both executions, and the final state is the same.
Proof as before.

2.2 Concrete partial reversal algorithm

BG give a concrete version of this algorithm that is claimed to be the same as the abstract version. I don't think this is exactly true—there seem to be some issues involving initialization.

Here is the concrete partial reversal algorithm:
Associate with each node u, at each time, a *height* triple (α_u, β_u, u). Here, α_u is a nonnegative integer, β_u any integer. Again, the triples are totally ordered, lexicographically. GB require that, initially, all the α_i values are 0 (but that doesn't make sense if the algorithm is supposed to start from arbitrary configurations). Edges are directed from higher to lower "heights" as before, which prevents cycles.

Now each node u, when it "fires", sets $\alpha_u := min_{v \in N_u} \alpha_v + 1$. (The full reversal algorothm used the max here, instead.) Also, it sets β_u to be smaller than β_v for any $v \in N_u$ such that α_v is equal to the new α_u. (To be specific, and to make the algorithm deterministic, take the minimum such value, -1.

Thus, for any neighbor for which the α's themselves don't resolve the order, orient the edge inwards towards u. The effect is for node u to insert itself "as low as possible" in the order, ensuring that at least one edge is directed outward, but otherwise, allowing everything possible to remain oriented inward.

Does this actually implement the "list" description above? This is not obvious. We have to consider 3 cases to see that the two versions of the algorithm make the same decisions about link reversal in both cases.

1. If the list is not full (is equal to N_u), then the concrete algorithm doesn't reverse the links to the listed neighbors.
2. If the list is not full, then the concrete algorithm does reverse the links to all neighbors not on the list.
3. If the link is full, the concrete algorithm reverses all the links.

Consider Case 1:
The list consists of neighbors v that have reversed (u, v) to point towards u since u last fired. Consider each such v. When it last reversed (u, v) to point towards u, it caused h_v to be set $> h_u$. Thus, it must have set $\alpha_v \geq \alpha_u$. Moreover, we claim that actually it set $\alpha_v > \alpha_u$: if they were set equal, v would have set the tiebreaker β_v to be $< \beta_u$, which would make $h_v < h_u$. In fact, it must be that $\alpha_v = \alpha_u + 1$—because v took u into account in performing its min calculation, taking the min + 1.

Thus, when u fires, every neighbor v of u "in the list" has $\alpha_v = \alpha_u + 1$.

Also, no neighbor v can have $\alpha_v < \alpha_u$, since that would represent an outgoing edge from u, and u is a sink when it fires. So, every neighbor v has $\alpha_v \in \{\alpha_u, \alpha_u + 1\}$.

We must argue (somehow) that, if a neighbor v isn't in the list, then $\alpha_v = \alpha_u$ when u fires. But why is this true? It seems to require an inductive argument, starting from an initial state in which all the αs are equal. This remains to be worked out.

Now, when u fires, it sets α_u to $\alpha'_u = min_{v \in N_u} \alpha_v + 1$. Since we are assuming in Case 1 that the list is not full, and each neighbor v not on the list has $\alpha_v = \alpha_u$, it must be that $\alpha'_u = \alpha_u + 1$, which is the same as the α_v values for all the neighbors that are on the list. Then, u uses β to ensure that,

for every v with $\alpha_v = \alpha'_u$, the edge (u, v) remains pointing inward towards u. That is, it doesn't reverse the edges from these nodes—the nodes on the list.

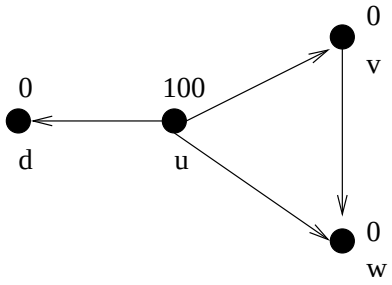
Consider case 2:
The reasoning in Case 1 also shows that (if the list isn't full) u does in fact reverse the edges to all the non-listed neighbors.

Consider case 3:
Assume the list is full, that is, all neighbors have reversed into u since u last fired. Then it seems that all will have $\alpha_v = \alpha_u + 1$. Then when u fires, it sets $\alpha'_u = \alpha_u + 2$, which does reverse all the edges, as specified in the abstract algorithm.

It seems we could prove some useful invariants, with GB's initial conditions, e.g.: If $v \in N_u$ then $|\alpha_u - \alpha_v| \leq 1$.

Exercise: Prove carefully that the concrete partial-reversal algorithm correctly emulates the abstract partial-reversal algorithm. You could use a simple simulation relation, and some simple invariants.

Thus, it seems to me that the correctness of the correspondence between the concrete and abstract partial link reversal algorithms depends on the particular kind of initialization used in BG—with the α's all initially equal. In fact, the behavior seems to be different if we remove this constraint on the initialization: Consider this example (see the figure): Consider nodes d, u, v, w, with edges



$(d, u), (u, v), (u, w), (v, w)$. α_d is initially 0, α_u initially 100, α_v and α_w initially 0. Thus, the initial edges are oriented from u towards d, w, and v, and (say) from v to w. w is a sink, so it fires, computing $\alpha_w = min + 1 = 1$. This reverses edge (v, w) but not edge (u, w) (since the list is empty, this is different from what would happen in the abstract algorithm, where both edges should be reversed). Then v is a sink, fires, computes $\alpha_v = 2$, reversing only one edge,... This goes on for around 100 reversal steps. At each such step, the behavior of the concrete partial reversal algorithm is different from that of the abstract one.

2.2.1 BST Analysis

They consider a version of the concrete algorithm in which the αs can be initialized arbitrarily. They justify this on practical grounds—because of network connections and disconnections, we can't really ensure particular relationships among neighboring αs. This does seem to make practical sense, and implies that BG missed something by relying on strong initialization assumptions. Really, what is interesting is stabilization from some arbitrary state that might be reached in a *changing* network, not just those that are reached in an unchanging one.

BST do not seem to notice that they are doing something different from BG here. In particular, they claim (on page 212) equivalence between their concrete version and the list-based version, but their "proof" is incomplete—and wrong.

However, if we consider just the concrete version of the algorithm, with their arbitrary initializion, they do get some interesting results: Say b is the number of initially-bad nodes. They get an $O(ba^* + b^2)$ work and time bound, where $a^* = max(\alpha) - min(\alpha)$, the discrepancy between the min and max values of α, in the initial state. They also show that this bound is tight for this algorithm, by exhibiting a bad execution.

Their upper bound proof proceeds by classifying the bad nodes into layers again, this time based on the shortest *undirected* path from the node to a good node. (Recall before we used the smallest number of edges that have to be traversed in the wrong direction.)

They prove a lemma bounding how large a node's level can get before it must become good:
Lemma 5.1: When a node u in level k becomes good, $\alpha_u \leq amax + k$. (This refers to the max α in the initial state.)
Proof: By induction on number of levels.

Since at each reversal, the α value of a node increases by at least 1, they get:
Corollary 5.2: A bad node reverses at most $a^* + b$ times before it becomes good. (b is the number of bad nodes). So, the total work (and so the time) is $O(ba^* + b^2)$.

They also show that these bounds are tight, by exhibiting graphs and initial settings that achieve them. The construction for the work bound is a little intricate, depending on the details of the link reversal rule. (Now the levels in the graph are arranged in pairs, where the number of reversals is the same for two consecutive levels.) The construction for time uses the same trick as in the full reversal case—inserting a clique with a lot of nodes at the last layer, where we know the nodes will need to perform many reversals. Since it's a clique, all the reversals must be done sequentially.

They conclude from all these results that the full reversal algorithm has better worst-case performance, since it's always $O(b^2)$—regardless of initial state setting. Q: Why would you then bother with the partial reversal algorithm?
A: It might very well be better on average.

2.3 The general class of algorithms

This is described only concretely, in terms of heights. It generalizes the full and partial reversal algorithms. Instead of just integers, pairs, triples, etc., they now assume that the node heights are chosen from some totally-ordered set A, with disjoint subsets A_u for all the nodes u. Links are directed according to the ordering of the heights (here, according to the ordering of A). Destination d has (initially) the smallest label, and never changes it.

The entire system state at any point is assumed to consist of an assignment of heights (in A) to all the nodes. The algorithm is assumed to be deterministic, in the sense that each node applies a function g_u when it fires—a function of the node u's state and the state of its neighbors. A step consists of one or more firings, by sink nodes.

They give three formal numbered assumptions:
(A.1) and (A.2) express the standard "safety" properties: Each node has a function g_u, and each step involves some sinks (at least one) applying their g_u functions. Also, the g_u functions are defined to increase the height of the node applying the function (according to the ordering on A).

Property (A.3) is more complicated. In fact, I think it is stated incorrectly. It's a sort of liveness property. What I think it's supposed to mean is: If we consider any execution of the algorithm subject to (A.1) and (A.2), in which some particular node u fires infinitely many times, then h_u is eventually larger than any particular element of A.
(What I think are some mistakes: they don't specify that we are talking about an execution satisfying the previous constraints. Also, this seems to count the increases that u would see if it applied g_u at every step where u was a sink, rather than just the steps where it actually happens to fire.)

They show that the full reversal and partial reversal rules (the concrete versions) are special cases—that is, they satisfy (A1)-(A3). For this generalization, they prove all the same claims we have already talked about:
Claim 1. If two nodes u and v both reverse in the same step, then u and v aren't neighbors.
Proof: As before, because both are sinks.

Claim 2. At each stage, the graph is acyclic.
Proof: This is because of the total ordering of heights.

Claim 3. If the graph is connected, then the algorithm eventually terminates, ending up with a destination-oriented DAG.
Proof: A proof like the second one I gave last time works—the one that argues that, if an execution doesn't terminate, then all the nodes must fire infinitely many times. But now, instead of arguing about lists and edge reversals, they argue about labels.

Thus, suppose that an execution doesn't terminate with a destination-oriented graph. Then there is always some sink, and so firing steps continue forever. Then there must be some particular node u that fires infinitely many times. It must raise its height each time. That means, by (A.3), that it raises its height beyond any A element, eventually. But, at each point where u fires, all of its edges are incoming. Now that means that the heights of its neighbors are all greater than its own. That in turn implies that all of u's neighbors must fire infinitely many times. Continuing this argument step by step in the (undirected) graph, we see that all nodes fire infinitely many times. But this can't happen for d, contradiction. QED Claim 3

Claim 4: If a node starts out having a directed path to d, then it will never reverse the direction of its links.
Proof as before.

Claim 5: Any two executions of the abstract partial reversal algorithm in a connected network, starting from the same global state (which includes not just link directions but also list information) are "equivalent": Each node performs the same number of reversals in both executions, and the final state is the same.
Proof as before.

BST has results for the general case:
They show that, for any arbitrary deterministic algorithm that fits the BG definitions, the time and work bounds are $\Omega(b^2)$. They use this to conclude that the full reversal algorithm is better, in the worst case. The construction is a similar layer construction to the one for the partial reversal algorithm, based on layers defined by minimum-length undirected paths to good nodes.

2.4 Advantages/disadvantages

Advantages:

- Basing the ordering on elements of a totally ordered set automatically prevents cycles.
- Since this method forms a DAG, not a tree, it can be used to maintain contingency routes in addition to primary routes.
- Asynchrony allows flexibility in when a node makes its local adjustments, in response to topology changes.

Practical implementation issues:

- The algorithm requires tight synchronization between neighbors, to make sure the link reversal happens atomically at both ends (that is, that the increase becomes known to the neighbors immediately). There is some work required to implement this atomicity.
- It would be nice to have a mechanism to allow resetting of large heights to low values.

Comparison with shortest-paths algorithms:
These algorithms don't generate shortest paths, just some paths. However, they have multiple routes; this allows some delay in responding to changes. Only nodes that lose all their paths to d need participate in the link-reversal algorithms, which means less communication overhead and more stability. They always guarantee loop-free routes (though not necessarily loop-free communication—messages can still be chasing around changing routes).

3 Park, Corson TORA algorithm

3.1 Introduction

They propose a link-reversal routing algorithm for real mobile wireless networks. They claim all the same advantages as GB: Adaptive, efficient, scalable, local responses to changes. They guarantee loop-free routes always. They also provide multiple routes: this alleviates congestion, and also means that many changes don't require any reaction at all. They establish routes quickly. There is low communication overhead; nodes maintain information only about neighbors. The reactions to topological changes are local.

They claim it's especially good for large, dense networks like typical mobile networks—presumably because such networks will have lots of redundant paths to destinations.

Their work adds two notable features:

1. It's designed to cope well with partitions, which they believe will be common in ad hoc networks. They argue that GB exhibits instability in portions of the network that get partitioned from the destination:
   Example: Two nodes may talk just to each other, separated from everyone else including d. Then they will keep alternately increasing their labels forever, for no purpose. This is communication-inefficient.

   TORA improves on GB by adding partition-handling; the protocol detects the partition and erases invalid routes.

2. It's on-demand, creating routes only when needed. Thus, edges can be undirected, as well as directed one way or the other; undirected means no routes pass through the edge. When new routes are needed, they are reestablished quickly.

They assume synchronized clocks, but claim that the algorithm would also work (perhaps not as well) with less synchronized clocks. They use clock values as parts of their height tuples, in fact, as the highest-order, dominating component. Hence, the name "Temporally-Ordered Routing Algorithm".

3.2 Notation and assumptions

- $G = (N, L)$, undirected.
- Nodes have ids.
- Bidirectional communication on all edges.
- Set L of links changes with time.
- Nodes may also fail (and recover?).
- Assume nodes know their neighbors (because of some lower-level protocol).
- They allow a delay in finding out about changes—they don't require atomic agreement on neighborhood changes as GB do.
- Assume all transmitted packets are received correctly and in order of transmission.
- Assume when u sends a message, it broadcasts to (received by) all neighbors in its set N_u.

3.3 Basic ideas

We focus on one destination d, as before. There are three functions: Creating, maintaining, erasing routes.

- *Creating routes:* A node u initiates establishment of a route to d only if all its adjacent links are undirected. In fact, route creation basically amounts to assigning directions to undirected links.
- *Maintaining routes:* When directed portions of the graph deviate from being destination-oriented, they do reversal steps to try to restore this property.
- *Erasing routes:* TORA attempts to detect partitions, and accommodate, by "clearing" all the links in the portion disconnected from d—that is, making them undirected again.

How exactly they do this is not so easy to follow.

They use 3 kinds of control packets:
QRY, query, for creating routes,
UPD, update, for creating and maintaining routes,
CLR, clear, for erasing routes.

They claim that TORA is actually a member of GB's general class.

TORA uses heights that consist of 3-part *reference levels* and 2-part *increments*. A sink node (local min) may select a new reference level, higher than any previous reference level that already exists anywhere in the network (a global max). It sets its height to a new height value containing that reference level. We call this a *generate step*.

This can cause other nodes to become sinks. Any such node can execute a partial reversal, with respect to their neighbors that already have the new reference level (don't reverse the edges to these nodes). This allows the new reference level to propagate through the network, but only extending through sink nodes (which are nodes that have lost all their routes). Call this a *propagate step*.

Any node that was a source before this reaction started (all its edges were outgoing) has to also adjust its height to make sure it remains a local maximum. For this purpose, it defines a higher sub-level associated with the new reference level (so its edges to those neighbors having the new reference level get directed outward). This new level is called a "reflected reference level". Call this a *reflection step*.

The reflected sublevels somehow get "reflected back" to the originator of the new level. If the originator gets it back from all its neighbors, it has determined that no route to the destination exists (a partition is detected). Then it can begin the protocol to erase the invalid routs. Call this a *detection step*. There is no proof that this works.

3.4 Detailed description

3.4.1 Data types

The labels are 5-tuples: $(\tau, oid, r, \delta, u)$, where: (τ, oid, r) represents the reference level, and (δ, u) represents the increment.

Reference level:

- τ is a time (e.g., based on synchronized clocks).
- *oid* is the id of the node that originated this reference level.
- r is a bit used to divide the reference level into (just) two sublevels; used to distinguish between the original reference level and its reflected version.

Increment:

- δ is an integer used to order nodes w.r.t. a common reference level; used in propagation of the reference level (?).
- u is the id of the node itself, a tiebreaker.

- h_u, $u \neq d$, height of node u, initially $NULL_u = (-, -, -, -, u)$.
- h_d, height of destination, always $ZERO = (0, 0, 0, 0, d)$.

Each node u also maintains $h_{u,v}$, representing its best knowledge of h_v for each neighbor v, initially $NULL_v = (-, -, -, -, v)$. Thus, they explicitly model the asychronous updating of the neighborhood information, by modeling the local knowledge of the neighbor sets.

Also, node u maintains a link-state array with entry $LS_{u,v}$ for each link (u, v), that is, it monitors the status of its own adjacent links.
$LS_{u,v}$ is:

- UP (directed inward) if both h_u and $h_{u,v}$ are non-$NULL$ and $h_{u,v} > h_u$.
- DN (outward), if both are non-$NULL$ and $h_{u,v} < h_u$; or if h_u is $NULL_u$ and $h_{u,v}$ is non-$NULL$.
- UN, if $h_{u,v} = NULL_v$.

Thus, the NULL nodes are considered upstream—their edges are directed outward. (The $NULL$ values don't seem to be explicitly ordered with respect to the other values...but the intuition seems to be that they are higher than the other values.)

New links can be activated by setting initial defaults as above.

3.4.2 Creating new routes

This uses QRY and UPD packets.
QRY: No info (just the destination name)
UPD: Height (label) of the originator of this UPD packet.

Each node maintains:

- RR, a flag saying whether a route is required or not, initially 0.
- time at which last UPD packet was bcast.
- time at which each adjacent link became active.

Each node performs the following:

- If u has no directed links, RR = 0, and u wants a route:
  - Bcast QRY; RR := 1.
- When node u receives QRY from v:
  - If u has no outgoing links then:
    —If RR = 0 then (not already looking for a route, flood the request): Bcast QRY; RR := 1.
  - If u has at least one outgoing link then (have found a route):
    —If $h_u = NULL_u$ then set h_u to minimum label of non-$NULL$ neighbors, adding 1 to the δ component.
    —Bcast UPD(h_u).
- When a link becomes active:
  - If RR = 1 then: Bcast QRY (that is, include the new link in the flooded search).
- When node u receives UPD from v:
  - Update $h_{u,v}$.

- If RR = 1 (which implies $h_u = NULL_u$) then (have found a route):
  —Choose a new height: Set h_u to minimum label of non-$NULL$ neighbors, adding 1 to the δ component.
  —RR := 0;
  —Bcast UPD(h_u).
- Update $LS_{u,v}$.

Basically, they seem to be flooding QRY packets until they reach a node that has a route. Then they update neighbors about latest heights.

3.4.3 Maintaining routes

Done only by nodes u with non-$NULL$ heights. There are several activities:

- *Become-NULL*:
  Precond: Node u has no directed links.
  Effect: Set $h_u := NULL_u$.
- *Generate*: Node u detects a failure and defines a new reference level.
  Precond: Node u has no outgoing links, due to a link failure, and has at least one incoming link.
  Effect: Choose a new h_u, with reference level $(t, u, 0)$, where t is the "time of the failure", and where $\delta = 0$.
- *Propagate*: Node u propagates the highest of its neighbors' reference numbers, but without reversing the edges incoming from those neighbors.
  Precond: Node u has no outgoing links, due to a link reversal following receipt of an UPD, and some neighbors have different reference levels:
  Effect: Choose a new h_u, with the highest of the neighbors' reference numbers, and with δ chosen to be smaller than the δ of all neighbors with that reference number. (This keeps the edges pointing inward, while propagating the highest reference number).
- *Reflect*: Node u generates the reflected version of a reference number shared by all of its neighbors.
  Precond: Node u has no outgoing links, due to a link reversal following receipt of an UPD, and the neighbors all have the same reference level, with $r = 0$.
  Effect: Choose a new h_u, with the same reference number, but with $r = 1$, and with $\delta = 0$.
- *Detect*: Node u detects a partition.
  Precond: Node u has no outgoing links, due to a link reversal following receipt of an UPD, and the neighbors all have the same reference level, with $r = 1$, and with $oid = u$. This reference level originated by u has been reflected and propagated back (with the higher sublevel $r = 1$) from all its nbrs. This is supposed to mean that a partition has been detected.
  Effect: Set $h_u := NULL_u$, and start route-erasure (see below).

In all cases, the node does obvious basic maintenance stuff like updating LS, broadcasting UPD backets whenever anything changes, and updating $h_{u,v}$ upon UPD receipt.

Whenever u loses an outgoing link that isn't its last, it just does basic maintenance but doesn't introduce new reference levels.

3.4.4 Erasing routes

When node i detects a partition (case 4 above), it "cleans up": sets its heights and neighbors' heights to NULL, updates LS, and broadcasts a CLR packet, which contains the reflected reference level.

When u receives CLR from v:
If the reference level in the CLR packet = u's reference level, it cleans up as above.
If it does not match, u sets the height for each neighbor with the same reference-level as the CLR packet to NULL and updates the LS entries correspondingly. (This helps clean up.) Also, in this case, if u thus loses its last outgoing link, it can now do the Generate step of maintenance.

Summary: When a failure causes a node u to lose its last outgoing link, it will re-establish a route in one pass of the set of nodes affected by the failure, if u is still connected to d. If not connected, u will detect the partition in 2 passes and all invalid routes will be erased in 3 passes.

Exercise: Can you turn this into time bound results?

3.5 Effects of time errors

Clocks are used to establish temporal order of the link loss events. Logical clocks could also be used to establish this order. What happens if the logical time order isn't the same as the real time order? They claim that the resulting algorithm is still in the BG general class. (Is this obvious? Maybe because we preserve all event dependencies, it doesn't matter...) Efficiency could be affected somewhat. They conclude that it's good to use clocks that are sufficiently synchronized so that the time between failures is much greater than the clock error.

3.6 Conclusions

They have no simulation results. But, they do have a table of claimed worst-case analysis results, for time and communication complexity of many algorithms. This deserves some study.

Possible improvements:

- Over time, the DAG may become less optimally directed than it was initially. (They claim that initially, the δs give (approximately?) the minimum distance (in hops) to d, but this can degrade with changes.)
- We could enhance the protocol by periodically propagating refresh packets outwards from d, to reset the reference level to 0 (?) and restore the meaning of the δs to be the shortest distances ("Periodic, destination-oriented route optimization"). Such a strategy could also support correction of corrupted state information (self-stabilizing?). Refreshes should occur at a low rate, in the background.