Reverse Path Forwarding

Christie Bolton and Gavin Lowe

October 12, 2003

Abstract In large networks nodes need to know the best path by which to route messages to other nodes. Hence, in large networks in which the topology of the system changes as links go up and down, efficient and reliable protocols are needed for determining such paths.

In this document we study the reverse path forwarding protocol for determining such paths. We use a variety of analysis techniques to identify necessary constraints and to prove correctness under these conditions, and we discuss the efficacy of each such technique. In particular we present counter examples found by the event-based model-checker FDR and the state-based Alloy Tool—part of our remit was to compare these tools—illustrating that the protocol does not work correctly if the cost of links is dependent upon the node using that link. We use a special-purpose network sampling program, which finds no inconsistencies after sampling a significant number of networks in which the cost of links is independent of the node using that link, thus increasing our confidence in the correctness of the protocol under these stricter conditions. Finally we perform a hand-proof to prove the validity of the protocol under these conditions.

The main contributions of this paper are therefore: the identification of an error in the reverse path forwarding protocol that can lead to nodes in a stable network having incorrect information about links in their connected component; a proof of the correctness of the protocol assuming additional constraints to overcome this error; a comparison of the model checkers FDR and Alloy.

Keywords: Routing; ad hoc networks; reverse path forwarding; analysis techniques; CSP; FDR; Alloy.

1 Introduction

In networks, nodes need to know the best path by which to route messages to other nodes. In order to do this, nodes exchange information about network links, so that the best route can be calculated.

We study here the use of a particular protocol, the *reverse path forwarding* routing algorithm (RPF) to distribute information about the state of network links. In this protocol, nodes continually broadcast information about the network. Each piece of information is time-stamped, and a node will update its routing tables only in response to receiving link-state information that is more recent than the currently stored information.

Each node n accepts a packet or message from node p that was initiated by node i only if n believes that p is the first node on n's shortest path to i; we will call p the *provider* to n for i in this case. We illustrate this in Figure 1.

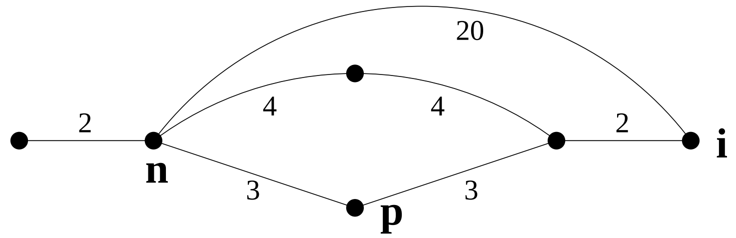

Figure 1: Node n accepts a packet initiated by i only from p since p is the first node on the shortest path from n to i.

It can be shown easily that under stable network conditions (i.e. where routing tables don't change), reverse path forwarding works correctly in the sense that a broadcast is accepted exactly once by every node in the originator's connected component, and the re-broadcasting eventually stops. However, as noted in [DM78], if the routing tables change, a broadcast message might never be accepted by a node.

We would like to show that if the state of the links eventually stabilises (i.e. they stop breaking and being remade), then the system will eventually stabilise into a state where every node has the correct information about every link in its connected component. We use a variety of tools and techniques to try to establish this result.

The paper begins with a brief discussion of established routing algorithms, both adaptive and non-adaptive, identifying their relative strengths and weaknesses. This is followed in Section 3 by a detailed description of the reverse path forwarding routing algorithm and its complexities, and a discussion of our choices of tools and techniques.

The following three sections discuss our tool-based analysis of the algorithm, firstly using FDR [Ros94] the model-checker for the process algebra CSP [Hoa85, Ros97], secondly using the Alloy Tool [JSS01], and thirdly using a special-purpose Java programme. In Section 7 we present a hand proof of the validity of the algorithm before concluding the document in Section 8 with a discussion of our findings. The code for the automatic analyses is included in the appendix.

2 Background

In any distributed network an efficient and reliable protocol is needed to pass packets between nodes. Various *routing algorithms* have been proposed to achieve this end. A routing algorithm should ensure that, provided the source and destination nodes are connected, each packet sent should eventually reach its destination. If the source and destination nodes are not directly connected then this will take multiple hops. Furthermore, the problem becomes more complicated if the network is *dynamic* and links may be made or broken.

There are two main types of routing algorithm: adaptive and non-adaptive. A *non-adaptive* or *static* routing algorithm is one in which the path on which a packet is sent is predetermined and hence unaffected by changes to the topology of the network or the number of packets being sent along particular routes. Nodes may or may not know the topology of the whole network but they must know their own neighbours—nodes to which they are directly connected—as well as the first node on a shortest path to each node to which they are connected.

In *adaptive* or *dynamic* routing algorithms, the path on which a node sends a packet is determined by that node's current perception of the state of the system, be it the topology of the network or the number of packets passing along each link. Beliefs about the state of the system are regularly updated as messages about changes in the state are propagated throughout the system. Hence the choice of path along which to send these messages and packets is also regularly updated.

2.1 Examples of non-adaptive routing algorithms

The *flooding* algorithm is a simple example of a non-adaptive routing algorithm. Every incoming packet that has not reached its destination is forwarded on every path except that on which it arrived. Although this protocol satisfies the condition that provided the source and destination nodes are connected a packet should eventually reach its destination, it is inefficient. The main drawback is that vast numbers of duplicate—and hence redundant—packets traverse the network. Indeed, unless certain *damping* measures are imposed, a potentially infinite number of messages can be generated for each packet sent. An example of such a damping measure is to record in the header of the message the number of times the packet has been forwarded and for the packet to be dropped if this exceeds the diameter of the network.

The *shortest path* routing algorithm [Dij59], is an example of a more efficient protocol for a static network. This algorithm requires that every node knows the topology of the whole network, or at least the subnet to which it is connected[1]. Each node constructs a graph of the network using a predetermined distance measure. It then forwards any incoming message to the first node on a shortest path to the destination node.

2.2 Disadvantages of non-adaptive routing algorithms

Although Dijkstra's shortest path routing algorithm is efficient and reliable for networks in which the topology never changes, there are many networks for which this assumption is too strong. Not only must we contend with the reality of machines crashing and links breaking, but current technology means that we need to consider networks in which even the geographical position of nodes may change.

When applying a non-adaptive routing algorithm we must be aware that if the state of the system changes, guarantees about the delivery of messages will no longer hold. In particular, if any link in the predetermined path between two nodes is broken then the packet will not be able to complete the path and will not reach its destination.

[1] If the topology of the whole network is not initially known to all nodes it can be learnt using the flooding algorithm as described above.

2.3 Examples of adaptive routing algorithms

The *distance vector* routing algorithm [Bel57, FF62], the routing algorithm originally used in the ARPANET [MW77], is an example of an adaptive routing algorithm. Every node maintains a table with two entries for each node to which it is connected: the first records its belief of the shortest distance to that node and the second records the first node on that shortest path. Every time unit, each node broadcasts to all its neighbours its current belief of the shortest distance between it and every other node, and all other nodes update their tables accordingly.

A disadvantage of the distance vector routing algorithm is that there can be a considerable time lag between a link breaking and all nodes in the network learning about that break. We illustrate this in Example 1 below.

Example 1 Consider the simple network illustrated in Figure 2. We ob-

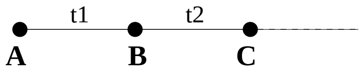

Figure 2: The distances AB and BC are respectively $t1$ and $t2$.

serve that the shortest distance to node A recorded initially by nodes B and C would be $t1$ and $(t1 + t2)$ respectively. Suppose now that the link between A and B breaks. In the first time unit after the break, B hears nothing from A but learns that the shortest path from C to A is of length $(t1+t2)$: B deduces that its shortest path to A must therefore be of length $(t1 + (2 \times t2))$. In the next time unit C learns that the shortest path from B to A is $(t1 + (2 \times t2))$ and deduces that its shortest path to A must be $(t1 + (3 \times t2))$. This pattern will continue with B and C alternately increasing their shortest path to A until some upper bound is reached and they both deduce that they are not connected to A. □

The distance vector routing algorithm was replaced in the ARPANET in 1979 by the *link state* routing algorithm [Per92]. In this algorithm each router maintains a map recording its current belief about the state of the network; whenever the state of a link changes—it comes up or goes down—this information is flooded throughout the network.

Broadcasting packets to the entire network eliminates the delay that occurs in the distance vector routing algorithm between a link breaking and nodes learning about the break: in Example 1 above, node C, hearing nothing from node A would immediately deduce that it was no longer connected to A and hence not feed false information back to B. However, as discussed in Section 2.1, flooding brings its own problems.

3 The Reverse Path Forwarding protocol and our chosen verification techniques.

The *reverse path forwarding* routing algorithm [DM78] is similar to the link state routing algorithm in that every node maintains a map recording its belief about the state of the network; every node determines the distance to each of its direct neighbours, and regularly distributes packets containing this information throughout the system. The main difference is that instead of using flooding, whereby the sender passes on messages to everyone at every hop with the recipients determining whether or not messages are redundant, in this algorithm some of that responsibility is put on the sender, thereby decreasing the number of messages sent.

More specifically, the originator of a message broadcasts it to all its neighbours. A node *id* will accept a message that it receives from node p and that originated from i only if p is the first node on the route that *id* would use to route messages to i; we'll call p the *provider* to *id* for i in this case. If routing is symmetric, then this is equivalent to saying that *id* normally receives from p messages that originate from i. If *id* accepts a message, it then broadcasts it to all its neighbours except p.

In our study we consider three simple structures of networks: a *chain*, a *ring* and a *complete graph*. The structures of these are as one would expect, and the graphs of variants of each are identified in Figure 3.

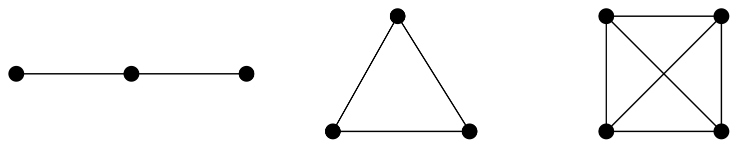

Figure 3: Graphs of a three-node chain, a three-node ring, and a four-node complete graph.

3.1 Complexity

Before discussing our choice of verification techniques, we consider the complexity of the system: why current model-checkers can consider only very small systems.

Since every link in the system can be either up or down, and assuming that knowledge about the state of any link can be either "old" or "recent"[2] it follows that each node can have any one of four beliefs about the state of any link. Hence, in a network with a possible n links, each node can have any one of 4^n beliefs. Moreover, there must be 4^{mn} belief sets in a network with m nodes and n links.

We see that the number of networks to be considered is exponential both in the number of nodes and the number of links. When considering a system as small as a four-node ring we need to consider the $4^{16} \approx 4 \times 10^9$ possible states that the network could be in.

3.2 Choice of verification techniques

There are a wide variety of tools and techniques available for automatically verifying the correctness of protocols. Model-checkers are typically excellent at identifying flaws or errors but can consider only finite systems. Other techniques should then be applied to prove the extensibility of results. Theorem provers, on the other hand, are very useful for proving results correct even for large or infinite systems. However, if the conditions are not quite right, theorem provers seldom shed light on how they should be adapted.

Part of our remit in this project was to compare the model checkers FDR and the Alloy Tool, associated respectively with the process algebra CSP and the state-based specification language Alloy, and so these were our first two choices of verification techniques. The results of these analyses are presented in Sections 4 and 5. A comparison of the two tools is presented in Section 8.4.

As observed in Section 3.1 above, the number of possible networks to be considered is exponential both in the number of nodes and the number of links, and even the most simple of systems will stretch current technology to its limits. Although both tools found an error—an inconsistency in beliefs between connected nodes about the status of a link in a stable network—in

[2]In Section 4 we discuss the simplifying assumption that knowledge is either "old" or "recent" and define formally what we mean by these terms.

the protocol for a small network, thereby indicating necessary additional constraints, even four node rings pushed them to the limit of their capabilities. To be confident that there were no errors for larger systems, even with these additional constraints, alternative proof techniques were necessary.

Before attempting a hand-proof we needed to be confident that our constraints were sufficient and so we used a specially built tool that generated large numbers of networks satisfying these constraints and checked for errors. This tool and the hand-proof are discussed in detail in Sections 6 and 7.

4 CSP-based analysis

In this section we discuss the CSP-based analysis of the reverse path forwarding protocol. We describe two models in the following two subsections; the models differ slightly in the way that nodes choose their providers.

- In the first model, we abstract away from the costs of edges, and suppose that each node id nondeterministically chooses the provider for each other node i from among those nodes j such that there is a path from id to i via j. It turns out that under these conditions, the protocol does not work correctly: we discover a network configuration where some nodes fail to find out about changes in the state of certain links. However, this arises from each such node choosing its providers in away that is arguably inconsistent, being incompatible with choosing routes on the basis of minimum cost.

- In the second model we arrange for each node to choose its providers so that messages are routed via a spanning tree. We again discover a network configuration where some nodes fail to find out about changes in the state of certain links. However, this configuration is not compatible with nodes choosing routes on the basis of minimum cost where those costs are the same for all nodes; however, it is compatible where different nodes can experience different costs from using the same links.

4.1 First CSP model

In the first model, each node id chooses providers for each other node i by picking a path to i along those edges that it believes to be up, and then setting the provider for i to be the first node on that path. Hence, the provider is chosen nondeterministically from all those nodes that could feasibly be a provider.

This description should be read in conjunction with the CSP script in Appendix A.

Messages The messages that are passed round the network represent beliefs about the state of the network. They are modelled by tuples of type:

$$Node.Node.Status.Age$$

Such a message $i.j.s.t$ represents that node i said t time units ago that the link from it to node j had status s (either *Up* or *Down*). We will talk more about the *Age* component later.

The model uses events of the form $send.id.k.i.j.s.t$ to represent node id sending such a message to node k.

Each node can signal its beliefs about the status of the links of the network; these beliefs are used by the specification process to test whether the system eventually stabilises into a state where every node has correct beliefs. An event $signal.id.\{i, j\}.s$ indicates that node id believes the link between nodes i and j has status s.

Time plays an important role in the protocol, so we record time passing by the event *tock*.

We could, in principle, have tagged each belief with the time at which it was recorded, instead of the age of the belief. However, if we had done so, we would have had to set an upper limit on the timestamps, to keep the state space finite. Instead, we record how old the belief is, up to some maximum age *maxAge* (so we do not distinguish between different messages once they reach a certain age). It turns out that it is sufficient to take *maxAge* to be 1: if the network is connected, it is always possible for a node to receive a message about a remote link with age 0, which will mean that old beliefs can always be replaced with more up-to-date beliefs.

Links Each link process has two states, corresponding to whether the link is made (i.e. working normally) or broken. When the link is made, it allows messages to be sent along it.

Each link has identity of the form $\{i, j\}$, for $i \neq j \in Node$, representing that it connects the two nodes i and j.

Node processes Each node process needs to store the following information:

- The status of local links, which can be detected directly;
- The believed status of remote links, based upon messages received from those links;
- The current *provider mapping*, mapping each other node to the provider for that node.

In order to reduce compilation time, each of these pieces of information is stored in a separate process, described below; these processes are combined together in parallel to give the desired behaviour.

Status of local links The node id includes a component $Local(id, j, s)$ for each other node j, keeping track of the status of the link from id to j; the parameter s records whether the link is currently up or down. This component synchronises with the corresponding *Link* process to detect the link going up or down. The process can send the status of the link to any other node, tagging the information as being of age 0. It can signal the state of the link. Finally, it can cooperate with the choosing of the provider mapping (described below).

Status of remote links The node id includes a component $Remote(id, i, j, s, t)$ for each non-local link $\{i, j\}$; the parameter s records the believed status of the link; the parameter t records how long ago that status was recorded. The process can send this belief to any other node. It can receive information about this link from another node; the *ChooseProvider* process, below, will ensure that such messages are received only from the appropriate provider; such messages are accepted only if the age is strictly less than the age of the current piece of information about the link. The process can allow one time unit to pass, and increment the age of the information; however, to keep the state space finite, the age is not incremented to more than the value $maxAge$. It can signal its belief about the status of the link. Finally, it can cooperate with the choosing of the provider mapping (described below).

Choosing and enforcing providers The node *id* includes a component *ChooseProvider*(id, i, p) for each other node i; the parameter p records the current provider for i (to id). The process is responsible for selecting the provider for i nondeterministically from the set of all feasible providers, i.e. from the set of those nodes j such that there is a path from *id* to i that starts off via j.

The process can perform an event *setProviderInt.ns* where *ns* is a sequence of nodes, with the first node *id* and the last node i, representing a path from *id* to i; it will synchronise with the *Local* and *Remote* processes such that this event will occur only if this node believes that all the links on that path are up, i.e. if for every consecutive j and k in *ns*, the link $\{j, k\}$ is believed to be up; the process then chooses the second node in *ns* to be the new provider for i.

Alternatively, the process can perform an event *disconnected.id.ns*, where *ns* is a set of nodes containing i, to indicate that there is no path to the nodes in *ns*, and in particular to i; in this case, the process sets the provider for i to be the null value 0. The process synchronises with the *Local* and *Remote* processes so that this event will occur only if this node believes that every link from a node not in *ns* to a node in *ns* is down.

The process can synchronise on messages sent by p about information that originated from i; thus it blocks all messages originating from i that are received from a node other than the provider, thereby ensuring that the *Remote* processes accept messages only from the appropriate providers.

Schedulers In order to reduce the size of the state space, the model of the system includes a global scheduler. The scheduler forces nodes to re-choose their providers in certain states, and allows other events to occur only when all nodes have chosen their providers. Initially, the scheduler forces each node i to choose its provider for each other node j, in increasing order of i and j. It forces each node to re-choose its providers after receiving a message, or after a local link changes state.

The system Each node is built up from a parallel composition of its components. The system is built from three nodes connected in a ring, together with a scheduler. All events are hidden other than those modelling links going up and down and the signalling of beliefs.

Specification The specification process $Spec(graph, instability)$ keeps track of the connectivity of the network in the parameter *graph* (a set of edges), by taking account of links making and breaking. The specification insists that once the network has stabilised—i.e. no more messages can be circulated—each node *id* should signal only true beliefs about the status of links that are in the same connected component of the graph as *id* (*id* cannot be expected to know about the status of links outside its connected component).

We consider, now, how quickly the network will stabilise after a link goes up or down. The network cannot be guaranteed to stabilise before the next *tock*, because there might be contradictory messages circulating about the link that has changed in the network, each with age 0. Even after a *tock*, the network might not have stabilised, because our model does not insist that nodes communicate whenever they can; however, such states where messages could be sent are also unstable in the CSP sense.

Our specification therefore initially allows any signals of beliefs; but if at least one *tock* has happened since the last change in the state of the network, then the system should stabilise (in the CSP sense) into a state where only true belief signals about links in the same connected component are available. The parameter *instability* records whether a *tock* needs to occur before the network can be guaranteed to be able to stabilise in this way.

Results FDR discovers that a system of three nodes fails to meet the specification. It finds that after the trace $\langle break.\{1, 2\}, tock\rangle$, the system refuses $signal.3.\{1, 2\}.Down$; i.e. node 3 does not realise that the link $\{1, 2\}$ has broken.

Using the FDR debugger to remove one layer of hiding, we find that the trace is as follows, where an event of the form $setProvider.id.i.p$ represents that *id* chooses p to be the provider for i; important events receive a comment.

$\langle$ *setProvider*.1.2.2	
setProvider.1.3.2	
setProvider.2.1.1	
setProvider.2.3.1	
setProvider.3.1.2	3 decides 2 is the provider for 1
setProvider.3.2.1	3 decides 1 is the provider for 2
break.$\{1, 2\}$	link $\{1, 2\}$ goes down
setProvider.1.2.3	
setProvider.1.3.3	
setProvider.2.1.3	
setProvider.2.3.3	
tock	time passes
send.3.2.3.1.*Up*.0	
setProvider.2.1.3	
setProvider.2.3.3	
send.3.1.3.2.*Up*.0	
setProvider.1.2.3	
setProvider.1.3.3 $\rangle$	the network is now stable

Note that node 3 will accept messages originating from node 1 only if they are received from node 2; and will accept messages originating from node 2 only if they are received from node 2. However, once the link $\{1, 2\}$ goes down, there is no path by which such messages can reach node 3, and so this node never accepts a message telling it that the link has gone down. This is depicted in Figure 4.

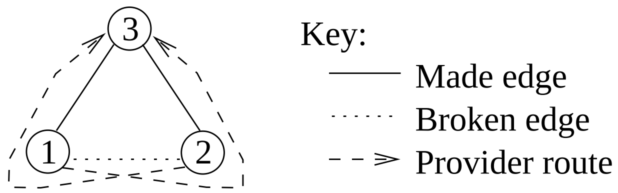

Figure 4: The error of the first system model

Discussion This model reveals an interesting aspect of the protocol. However, the circumstances under which it fails are perhaps unrealistic: it seems unlikely that node 3 would ever choose 1 as the provider for 2, and 2 as the provider for 1; for example, if routing paths are chosen based upon minimal costs, then there is no choice of positive costs for edges consistent with this routing decision.

It seems that we have abstracted too much detail from the way in which nodes choose their providers: the providers (to a particular node *id*) for different nodes are chosen independently, whereas we would expect there to be some correlation between them. The model in the next section seeks to overcome this over-abstraction.

4.2 Second model

In the second model, each node *id* chooses *all* its providers in a single event. More precisely, it chooses a spanning tree of those nodes in its connected component, and chooses the provider for each such node i to be the first node on the path to i in the spanning tree.

Much of the model is very similar to the previous model, so we describe only the differences. The complete script can be found in Appendix B.

For reasons of efficiency, this model does not include explicit link processes: instead, nodes i and j synchronise on events representing the link $\{i, j\}$ going up or down, so that each node knows the state of local links; the model of the node is such that it will not send messages along a link that is currently down.

As mentioned above, each node *id* chooses its providers by forming a spanning tree of those nodes in its connected component. This is represented by a event of the form *setProviderInt.tree*, where *tree* (a set of edges) is chosen from the set of trees including *id*. The *Local* and *Remote* processes synchronise on this event, to ensure that *tree* contains only links that are up, and to ensure that every link between two nodes, one in *tree* and one not in *tree*, is down, thereby forming a spanning tree for as much of the network as possible.

Results When we model a network of three nodes in a ring, as in the previous section, FDR confirms that the specification is satisfied.

However, when we model a network of four nodes in a ring, FDR finds an error. It finds that after the trace $\langle break.\{2,3\}, tock\rangle$, the system refuses $\{signal.4.\{2,3\}.Down, signal.1.\{2,3\}.Down\}$; that is, the arc $\{2,3\}$ has broken, but neither nodes 1 nor 4 realise it.

Using the FDR debugger to remove one layer of hiding, we find that the trace is as follows, where an event of the form *setProviders.id* represents node *id* choosing its providers; in important cases, we show the corresponding *setProviderInt* event (from which the *setProviders* event is obtained by renaming), so as to show the provider tree chosen.

$\langle$ *setProviders*.1	
setProviders.2	
setProviders.3	
setProviders.4	
break.$\{2,3\}$	link $\{2,3\}$ goes down
setProviders.2	
setProviders.3	
tock	time passes
send.1.2.1.4.*Up*.0	
setProviders.2	
send.4.1.4.3.*Up*.0	
setProviders.1	obtained from *setProviderInt*.$\{\{3,4\},\{1,4\},\{2,3\}\}$
send.1.2.4.3.*Up*.0	
setProviders.2	
send.1.4.1.2.*Up*.0	
setProviders.4	obtained from *setProviderInt*.$\{\{1,4\},\{1,2\},\{2,3\}\}$
send.4.3.1.2.*Up*.0	
setProviders.3	
send.4.3.4.1.*Up*.0	
setProviders.3 $\rangle$	the network is now stable

Note that node 1 will accept messages from nodes 2 or 3 only if they are received from node 4; and node 4 will accept messages from nodes 2 or 3 only if they are received from node 1. Thus, neither node 1 nor 4 will ever accept a message from node 2 or 3 informing them of the state of the link $\{2,3\}$. This is depicted in Figure 5.

Discussion One might ask whether the routing, above, could ever occur. We believe that it could: suppose the cost of each edge depends upon who the message is for. If the edge 1–2 is very expensive for node 1, and the edge 3–4 is very expensive for node 4, then they will route in this way. What this scenario shows is that if the cost of each edge depends upon who is using that edge, the protocol will not always work.

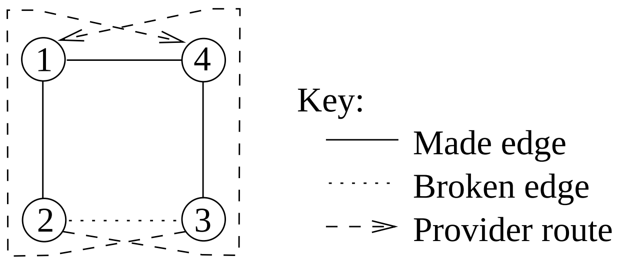

Figure 5: The error of the second system model

This raises the question as to whether the above scenario could arise in a setting where the cost of each edge is the same for all nodes. It is possible to get into a state where the nodes' beliefs about the costs of the edges differ, as above, and so the nodes would route as above. However, such beliefs would be unstable: nodes 1 and 4 would exchange messages about the true costs of edges, so that the beliefs would be updated.

In principle it would be possible to build a CSP model in which nodes calculate routing trees based upon shortest paths. However, we don't believe this would really be feasible, as the current models are pushing the limits of what FDR can cope with.

5 Alloy-based analysis

In this section we discuss the Alloy-based [JSS01] analysis of the reverse path forwarding protocol. This description should be read in conjunction with the Alloy scripts in Appendices C and D.

5.1 Features common to both models

During the course of our analysis we considered two models which differ only in the measure used to determine the distance between nodes. Since the main body of these models is so similar we introduce first their shared features before discussing their differences. We then conclude the section with a discussion of the results obtained in the automatic analysis of each.

Type declarations Before formally defining the main data types within our models, we introduce some basic types. The type *Configuration* can take three values: *Ring*, *Chain* and *CompleteGraph* representing the three topological structures under consideration.

Each *vertex* within the graph is associated with a unique identifier of type *NodeId* which we use to track the changes in the state of the node at that vertex. Each *edge* is represented by an element of type *Link* that records its identifying *ends*, a set of two *NodeId*s. A *TimedLink* records both a link and a timestamp of type *Age* that indicates how up-to-date the information about that link is. As with the CSP models it is sufficient to have only two ages: *Recent* and *Old*. Nodes will always have recent—and correct—knowledge about the status of *adjacent* links, that is links originating at the node. Furthermore, whenever a link is made or broken, all beliefs about the status of *non-adjacent* links will become old.

We consider three types of operation: *Propagate* which represents the passing of recent information between nodes; *Make* which represents the making of a line and records the associated line; and *Break* which represents the breaking of a line and also records the associated line. These are all of type *Op*. A sequence of such operations[3] is of type *Trace*.

The final basic type is the measure *Distance* which is used to determine the distance between nodes and will be discussed in greater detail in Section 5.2.

NodeState The data type *NodeState* modelling the current state of a node has six components. The first component, *id* : *NodeId*, identifies the vertex with which the node is associated. The next two components, *believesUp* and *believesDown*[4], are sets of timed links. Together these components record the node's current beliefs about the status of each link in the network along with the age of that belief: if, for instance, a node has recent knowledge that a particular link is up then the *TimedLink* recording both the link and the timestamp *Recent* will be included in the set *believesUp*. Constraints ensure that each node must have a belief about the status of every link in the

[3]Although a sequence is not a standard structure of Alloy an equivalent structure can be constructed easily.

[4]For simplicity we include the set *believesDown* in our description of a node but the implementation does not rely on this: our model is sufficient for networks in which nodes record their beliefs about which links are up but do not know the topology of the whole system.

network, but that no node can simultaneously believe a link to be both up and down. Furthermore, no node can have both old and recent information about the status of a given link.

The component *believesConnectedTo* is a set containing the identifiers of the vertices that the node believes it is connected to. The value of this component is determined by the set *believesUp*.

The remaining two components of a node's state space are both functions. The function *distance* is of type $NodeId \nrightarrow Distance$. Its domain is the set *believesConnectedTo* and for each node identifier in this set it returns, according to the chosen measure, the distance to that node. We discuss this further in Section 5.2. The final component is the function *onPathTo* which is of type $NodeId \nrightarrow NodeId$. It also has the set *believesConnectedTo* as its domain, and for each node identifier in this set it returns the identifier of its provider. This is calculated using the *distance* function and the set *believesUp*.

Network The data type *Network* records a snapshot of the system at any given time, and records both global and local information. The local information is contained in the set *nodes*, a set of *NodeState*s, one for each vertex in the graph, as described above. A network has a component *config* of type *Configuration* that records whether the graph is a *Chain*, a *Ring* or a *CompleteGraph*. The set of links *maximalGraph*, which is determined by the configuration of the system and the number of vertices, records all possible links in the network, whereas the subset *upLinks* contains only those that are up. Observe that this is a global variable and may contradict the beliefs of any of the nodes. The final component of a network is the *trace*. This records the sequence of operations that led to the network being in its current state.

Operations on the network We have already observed that there are three operations on the state: the making of a link, the breaking of a link and the propagation of recent information.

The precondition of breaking a link is that there must be a link to break. After the link has been broken, the nodes at either end both have recent knowledge that the link is down. All other beliefs about the status of links remains unchanged except that recent information about each link not adjacent to the node becomes old.

Consider, for instance, the four-node ring illustrated in Figure 6. Initially

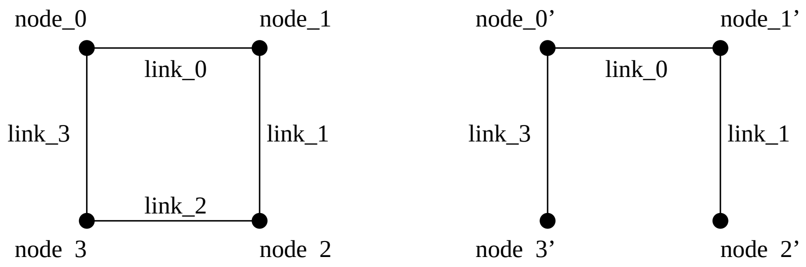

Figure 6: The link *link_2* is broken in a four-node ring in which all links are initially up and all nodes have recent knowledge that all links are up.

all links are up, and suppose that all nodes have recent knowledge that all links are up. If the link *link_2* breaks then the adjacent nodes, *node_2′* and *node_3′*, will both have recent knowledge that the link is down whilst the non-adjacent nodes, *node_0′* and *node_1′*, will continue to believe that *link_2* is up but will mark their beliefs with the timestamp *Old*. In addition: *node_0′* will have recent knowledge that *link_0* and *link_3* are up and an old belief that *link_1* is up; *node_1′* will have recent knowledge that *link_0* and *link_1* are up and an old belief that *link_3* is up; *node_2′* will have recent knowledge that *link_1* is up and old beliefs that *link_0* and *link_3* are up; and finally, *node_3′* will have recent knowledge that *link_3* is up and old beliefs that *link_0* and *link_1* are up.

The precondition and postcondition of making a link are dual to those of breaking a link. The precondition is that there must be a link to make. The postcondition is that the nodes at either end both have recent knowledge that the link has been made, and all other beliefs about the status of links remains unchanged except that for non-adjacent links recent information becomes old.

The precondition of propagating information is that there must be some information to propagate. In other words, there must be two nodes *id* and *p* and a link *link_0*, as illustrated in Figure 7 such that *id* has old information

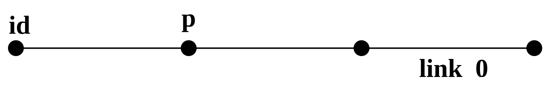

Figure 7: *node_p* is the provider to *node_id* for both ends of *link_0*.

about *link_0*, p has recent information about *link_0* and p is *id*'s provider for at least one of the nodes at the ends of *link_0*. If this is the case, then *id* updates its belief about the status of *link_0* to p's belief about that status, and records this information as recent. The state of all other nodes remains unchanged. One or more pieces of recent information may be propagated in a single *Propagate* operation.

WorldView Having observed that nodes record local information and that networks record both local and global information and provide a snapshot of the system, we now present our final data type, *WorldView*, which describes how the system has evolved.

A *WorldView* records the relations on networks *makeLink*, *breakLink* and *propagate* that respectively model the making of a link, the breaking of a link, and the passing of information between nodes. It records also networks *initial* and *final*. The state of the network *final* must be reachable from the network *initial* by applying some sequence of operations—making links, breaking links and propagating information—and this sequence will be recorded in its trace component. Our model imposes certain restrictions on the *initial* network: all links must be up and all nodes must have recent knowledge that all links are up. The effect of this is simply to reduce initial non-determinism and hence the size of the state space.

An abstraction we employ in both our models is that, rather than nodes broadcasting their view of the system at regular intervals, a pre-condition of the propagate operation is that there is some information to propagate. The resultant effect is equivalent, since nodes discard redundant information.

5.2 The distance measure

As observed previously, the only difference between our two Alloy models of the reverse path forwarding algorithm was the distance measure used to determine the shortest path between nodes. Obviously this is a critical part of the model since it is then used to determine providers for each node.

The first model In the first model (see Appendix C), our distance measure was simply the number of links between nodes. Using this measure, in the 4-node ring illustrated in Figure 8, *node_1* would be chosen as its

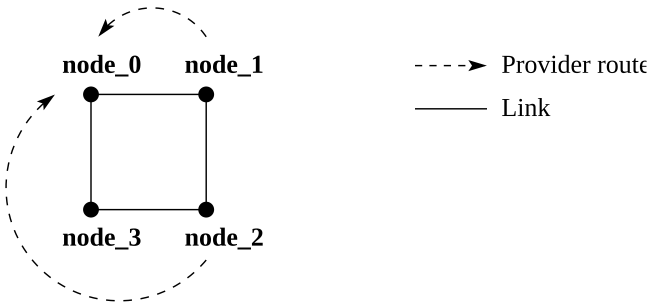

Figure 8: The providers to *node*_0 for nodes *node*_1, *node*_3 and *node*_2 are respectively nodes *node*_1, *node*_3, and either *node*_1 or *node*_3.

own provider to *node*_0 as would *node*_3, and non-deterministically, either *node*_1 or *node*_3 might be chosen as the provider to *node*_0 for *node*_2.

The second model In the second model (see Appendix D), our distance measure is much less restrictive. We insist that each node *id* calculates providers by picking a spanning tree of what it believes to be its connected component, and then for each node j, chooses the provider for j to be the first node on the path from *id* to j.

Observe that under this measure, a node need not be a provider for itself to a second node even if there is a direct link between the two nodes. This is illustrated in Figure 9 where *node*_1 has been selected as the provider to *node*_0 for *node*_3 (and for *node*_1 and *node*_2).

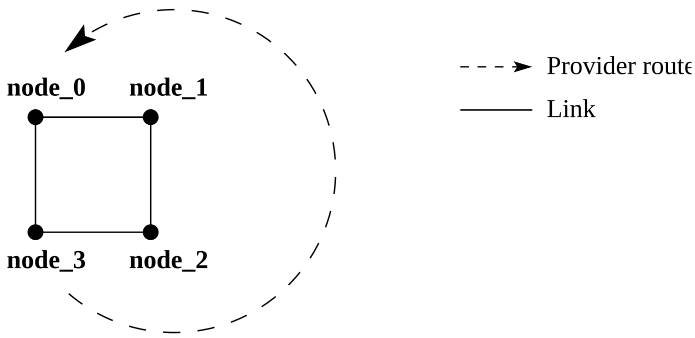

Figure 9: *node*_1 is the provider to *node*_0 for *node*_3.

5.3 Results

For both models we used the Alloy Tool to search the state space for a network that is reachable from the initial state in which there was no more information to propagate but two connected nodes had conflicting beliefs. In each case all links were up and all nodes had recent information that all nodes were up in the initialised networks. We considered chains and rings with up to four vertices. The state space was too large for complete graphs even for systems with only four vertices.

First model In the first model no counter-example was found for up to four nodes in a ring and up to three operations.

Second model In the second model, the Alloy Tool found a counter example in the case of a ring with four vertices. This example, illustrated in Figure 10, was analogous to that found by FDR and discussed in Section 4.

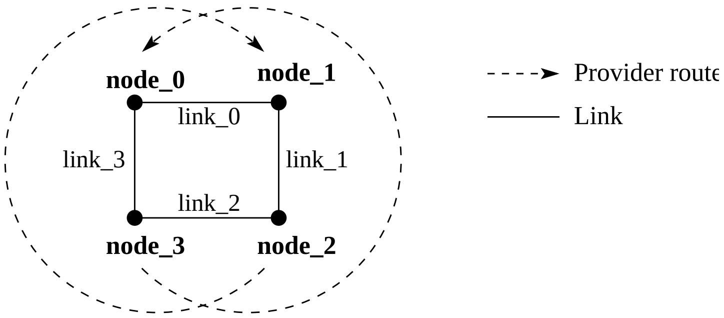

Figure 10: The counter example for alloy model 2.

Initially providers were chosen as follows:

Node	*Node*	*Provider*	*Node*	*Node*	*Provider*
node_0	*node_1*	*node_1*	*node_2*	*node_1*	*node_1*
	node_2	*node_1*		*node_0*	*node_1*
	node_3	*node_1*		*node_3*	*node_1*
node_1	*node_0*	*node_0*	*node_3*	*node_0*	*node_0*
	node_3	*node_0*		*node_1*	*node_0*
	node_2	*node_0*		*node_2*	*node_0*

The first operation to occur in this counter example was that *link_2* broke. Hence *node_2* and *node_3* recorded this as recent information as well as changing the timestamps on their beliefs about non-adjacent links. Nodes *node_0* and *node_1* also changed the timestamps on their non-adjacent links. Hence the beliefs of the nodes were as follows:

Node	*Link*	*Belief*	*Age*	*Node*	*Link*	*Belief*	*Age*
node_0	*link_0*	Up	Recent	*node_1*	*link_0*	Up	Recent
	link_1	Up	Old		*link_1*	Up	Recent
	link_2	Up	Old		*link_2*	Up	Old
	link_3	Up	Recent		*link_3*	Up	Old
node_2	*link_0*	Up	Old	*node_3*	*link_0*	Up	Old
	link_1	Up	Recent		*link_1*	Up	Old
	link_2	Down	Recent		*link_2*	Down	Recent
	link_3	Up	Old		*link_3*	Up	Recent.

The second operation recorded the propagation of the following information: *node_1* told *node_2* that *link_0* is up; *node_0* told *node_3* that *link_0* is up; *node_1* told *node_0* that *link_1* is up; and *node_0* told *node_1* that *link_3* is up. The third operation then recorded the propagation of the following information: *node_1* told *node_2* that *link_3* is up; and *node_0* told *node_3* that *link_1* is up. This left the nodes with the following beliefs:

Node	*Link*	*Belief*	*Age*	*Node*	*Link*	*Belief*	*Age*
node_0	*link_0*	Up	Recent	*node_1*	*link_0*	Up	Recent
	link_1	Up	Recent		*link_1*	Up	Recent
	link_2	Up	Old		*link_2*	Up	Old
	link_3	Up	Recent		*link_3*	Up	Recent
node_2	*link_0*	Up	Recent	*node_3*	*link_0*	Up	Recent
	link_1	Up	Recent		*link_1*	Up	Recent
	link_2	Down	Recent		*link_2*	Down	Recent
	link_3	Up	Recent		*link_3*	Up	Recent.

Observe first that no more information can be propagated since the only old beliefs are those of *node_0* and *node_1* about the status of *link_2*, and those nodes will only accept information from each other. Moreover, there is a contradiction in that *node_0* and *node_1* (incorrectly) believe that *link_2* is up whereas *node_2* and *link_3* (correctly) believe that it is down.

6 A special-purpose Java program

We now consider reverse path forwarding in a setting where the costs of edges are independent of the nodes that use those edges. What we want to know is whether, under constant link states, the network will eventually stabilise into a state where every node has correct information about every edge within its connected component. In order to investigate this question, we wrote a special-purpose Java program to search for stable network configurations—i.e. where beliefs do not change—where nodes hold incorrect beliefs.

Note that in a stable network configuration, each node's belief about the edge (i, j) will be the same as the beliefs of its providers for i and j. We call this condition the *provider belief consistency condition*.

What we want to do is search for a network configuration where the provider belief consistency condition is satisfied for all nodes, but where some such beliefs are incorrect.

The program in Appendix E repeatedly generates random graphs, and then tries to generate stable beliefs for the nodes; if it succeeds, it tests whether any beliefs are incorrect.

For simplicity, the program assumes that the costs of all edges are equal, so it simply records whether an edge is made or broken. (Earlier versions of the program allowed edges to have different lengths, but this does not appear to give any advantage.)

Most of the work is done by the function *tryGraph*:

- It begins by generating a random graph, where each edge is made with probability *madeProb* ($= 0.3$).

- It then checks that the graph is connected, giving up on this graph if not.

- It initialises beliefs of nodes so that they have correct beliefs about local edges, but such that beliefs about non-local edges are set to the special value *UNKNOWN*.

- The program then systematically replaces such *UNKNOWN* values with the hope of finding a set of beliefs satisfying the provider belief consistency condition. For each node k in turn:

  - it replaces *UNKNOWN* beliefs by randomly chosen ones;

- it then calculates the providers for k based upon those beliefs (using the function *getProvider*, which calculates shortest paths using Dijkstra's algorithm);
- it then seeks to choose beliefs for those providers so as to satisfy the provider belief consistency condition: if the relevant provider's belief is *UNKNOWN*, it is set to the same as k's belief; otherwise, it tests whether the provider's belief agrees with k's, giving up on this graph if not.

- Finally, it tests whether any pair of nodes have beliefs that differ in some place, printing out the details of the graph if so.

The main function repeatedly calls *tryGraph*, printing out a running count of the number of graphs tried, until a graph satisfying the condition is found.

However, the program failed to find such a network, even after trying more than 200,000,000 networks. This led us to suspect that no such network exists, a fact that we prove in the following section.

7 A proof of reverse path forwarding with uniform costs

In this section we prove that if the costs of edges are independent of the nodes that use those edges, then under constant link states conditions, the network will eventually stabilise into a state where every node has correct information about every edge within its connected component.

We will use the following definitions.

Definition 1 A piece of information held by a node about an edge is *recent* if it is timestamped strictly more recently than the last time at which a link state changed. □

Note that recent information is necessarily correct.

Definition 2 A node n has d-*stabilised* if for every edge e that has at least one node within distance d of n, n has recent information about e. □

Definition 3 A network has d-*stabilised* if every node has k-stabilised. □

Proof Let S be the set of distances between nodes in the network:

$$S = \{d(n, n') \mid n, n' \in \mathit{Node}\}.$$

Order S as

$$S = \{d_0, d_1, \ldots, d_N\} \qquad \text{where} \qquad d_0 < d_1 < \ldots < d_N.$$

We show that, assuming no further link state changes, the network will eventually d_k-stabilise, for each k; the proof is by induction on k.

As the base case, for $d_0 = 0$, note that the network will 0-stabilise one time unit after a link state change.

Note also that once the network has d_k-stabilised, it will remain d_k-stabilised assuming no further changes in link state: a node will never overwrite recent information with non-recent information because of the way timestamps are treated.

We need to show that if the network d_k-stabilises, it will eventually d_{k+1}-stabilise, assuming no further link state changes. So suppose the network has d_k-stabilised, and suppose node x is at distance d_{k+1} from node n. We need to show that n eventually accepts recent information about all edges incident to x. Note that n knows a correct shortest path to x, by definition of d_k-stability: all such paths will be such that every edge has at least one node within distance d_k of n. Hence n will pick a provider p for x. We perform a case analysis over p:

- Case $p = x$: then n receives and accepts recent information about edges incident to x directly from x.

- Case $p \neq x$: then p is at distance at most d_k from x[5], so holds recent information about all edges incident to x, by definition of d_k-stability; p will then pass this information to n.

Hence n will eventually accept recent information about all edges incident to x, and so the network will eventually d_{k+1}-stabilise.

Hence, by induction, the network will eventually d_N-stabilise, where d_N is the diameter of the network. Hence every node will eventually hold recent information about all edges in its connected component. □

[5]This step uses the assumption that the costs of each edge are the same for all nodes; if this assumption does not hold, the cost of p sending to x might be greater than the cost of n sending to x via p.

8 Discussion

Our remit in this project was both to explore and prove the correctness of the reverse path forwarding algorithm, and to compare state-based and event-based model checkers, the Alloy Tool and FDR. Our measure of correctness was that in every stable state of the system no two connected nodes could have different beliefs about the status of a link in their connected component.

8.1 Automatic analysis of small systems

We began by exploring various interpretations of the algorithm for small networks using FDR and the Alloy Tool. These interpretations, and the results obtained for each were as follows:

CSP Model 1 Each node non-deterministically chooses the provider for each other node in its connected component.

An error is found when considering a three node ring: this occurs if node i chooses node p as its provider for q and node q as its provider for p.

CSP Model 2 Each node chooses all its providers in a single event: it chooses a spanning tree of those nodes in its connected component, selecting as the provider for each node n the first node on the path to n in the spanning tree.

This model overcomes the error identified by CSP Model 1 since no node can choose two nodes to be each other's provider. However, a different class of error is found when considering a four node ring: this occurs if node i chooses node j as its provider for nodes p and q whilst j chooses i as its provider for p and q.

Alloy Model 1 Each node uses Dijkstra's shortest path algorithm to determine the distance between nodes where each link is of unit distance. Providers are then the first node on a shortest path.

No errors were found in this model for networks up to a four node ring.

Alloy Model 2 Each node constructs a spanning tree that is a subgraph of its belief about the state of the system and then uses this to determine the providers for all nodes to which it is connected.

This model is equivalent to CSP Model 2, and the same error is found.

Validity of identified errors No errors were found in Alloy Model 1 in which links were taken as unit cost. However, this seems more restrictive a condition than we would wish. Regardless of whether the cost of sending a message along a link is financial, temporal or determined by some other measure, it is very unlikely that the cost of each will be identical.

At the other end other scale, CSP Model 1 appears to be unnecessarily non-deterministic. We could not think of a scenario in which a single node would sensibly choose to select two nodes as the providers for each other.

The scenario considered in both CSP Model 2 and Alloy Model 2 however seemed much more realistic. We came up with a variety of situations in which the cost of each link might be dependent on the originator of the message: for instance, in the telecommunications industry it might be more expensive for a subscriber of one network to send a message via a particular route than for a subscriber of another network.

Necessary additional constraints Given that the scenario considered in both CSP Model 2 and Alloy Model 2 is a plausible one, we assert that the error identified by both tools is a *genuine error* in the protocol. We then infer that in order to guarantee correctness, it is necessary to prevent the cost of links from being totally independent of the sender.

8.2 Automatic sampling of networks

Having identified an error in the protocol for networks in which the cost of a link was dependent on the originator of the message, our next step was to try to prove that there were no errors for networks in which the cost of a link was independent of the originator of the message.

Unfortunately we had already stretched our two model checkers to their limits. So, wishing to increase confidence in the correctness of these conditions before attempting a hand-proof, our next strategy was to use a specially written Java tool to sample a large number of such networks.

The special-purpose Java program operated in a somewhat different way to the model checkers, generating random network configurations, and testing whether they broke the protocol. Such an analysis is not exhaustive—indeed, there are approximately 3.5×10^{13} different 10-node networks, for instance, so an exhaustive analysis does not seem feasible. However, it is hoped that failing configurations are sufficiently dense in the space of all configurations that this technique would have hit one had there been one. Further, the analysis is faster than with FDR or Alloy, because it can immediately start analysing networks, whereas the states of interest in the FDR and Alloy models are situated quite deep within the search tree, and so are only reached after considerable analysis, probably only after an infeasible amount of analysis for networks with more than four or five nodes.

After a significant number of checks—over 2×10^8 networks of varying sizes and configurations were considered—we were sufficiently confident in the correctness of our conditions to attempt a hand proof.

8.3 Hand proof

Having found a flaw in the protocol for a small networks using our two model checkers and used this result to propose additional necessary conditions, and having increased our confidence in the sufficiency of these additional constraints using the specially built sampling tool, we were then ready to attempt a hand proof.

The hand proof confirmed our hypothesis that the reverse path forwarding algorithm is correct for networks in which each node calculates routes based on shortest paths, and uses these to determine the providers for all nodes in its connected component, and in which the cost of each link is independent of the sender.

8.4 A comparison of FDR and the Alloy Tool

This case study was a good one for comparing the capabilities of FDR and Alloy as it was equally natural to consider the problem from either a state-based or an event-based perspective: it was relatively simple to construct models of the system in both CSP and Alloy.

FDR and the Alloy Tool were well matched in this task: both found the same error for the four node ring, and both were unable to consider any larger system. Our only reason for favouring one tool over the other is that for each network the Alloy Tool took significantly longer than FDR either to find an error or to confirm that no such error exists. Perhaps this discrepancy is due to our relative lack of experience in coding in Alloy.

8.5 Conclusions

Assuming the protocol to be correct if all connected nodes have the same beliefs about all links in their connected component, our conclusions are as follows:

- For networks in which each node calculates routes based on shortest paths, and uses these to determine the providers for all nodes in its connected component:
  - the reverse path forwarding algorithm is *incorrect* if the cost of links can depend upon the node using that link;
  - it is *correct* if the cost of each link is independent of the sender;
- FDR and the Alloy Tool were comparable in that both found the same flaw in the protocol and both were unable to cope with networks larger than a four node ring;
- This example is pushing the limits of what is feasible.

Finally, none of the computer-based analyses was able to give us a guarantee of correctness, for which a hand-proof was necessary. In these days of powerful analysis tools, it is easy to become reluctant—or even scared—to perform proofs; however, sometimes a hand-proof is appropriate. The computer-based analyses can give us two things:

- They can help us to identify conditions under which the system can fail and hence constraints necessary for correctness;
- They can give us some confidence in the correctness of the system, which makes it worth while attempting a proof.

References

[Bel57] R. E. Bellman. *Dynamic Programming.* Princeton University Press, 1957.

[Dij59] E. W. Dijkstra. A note on two problems in connexion with graphs. *Numerical Mathematics*, 1:269–271, 1959.

[DM78] Yogen K. Dalal and Robert M. Metcalfe. Reverse path forwarding of broadcast packets. *Communications of the ACM*, 21(12):1040–1048, 1978.

[FF62] L. R. Ford, Jr. and D. R. Fulkerson. *Flows in Networks*. Princeton University Press, 1962.

[Hoa85] C. A. R. Hoare. *Communicating Sequential Processes*. Prentice Hall, 1985.

[JSS01] Daniel Jackson, Ilya Shlyakhter, and Manu Sridharan. A micromodularity mechanism. In *Proceedings of the ACM SIGSOFT Conference on Foundations of Software Engineering/European Software Engineering Conference (FSE/ESEC '01)*, 2001.

[MW77] John M. McQuillan and David C Walden. The arpa network design decisions. *Computer Networks*, 1977.

[Per92] Radia Perlman. *Interconnections: Bridges and Routers*. Addison-Wesley, 1992.

[Ros94] A. W. Roscoe. Model-checking CSP. In *A Classical Mind, Essays in Honour of C. A. R. Hoare*. Prentice-Hall, 1994.

[Ros97] A. W. Roscoe. *The Theory and Practice of Concurrency*. Prentice Hall, 1997.

A First CSP model

```
-- Reverse Path Forwarding, combined with routing information

N = 3 -- number of nodes

-- Type of nodes:

Node = {1..N}
other(i) = diff(Node, {i})

-- Type of links:

LinkSet = { {i,j} | i <- Node, j <- other(i) }

-- status of channels

datatype Status = Up | Down

otherStatus(s) = if s==Up then Down else Up

-- timing stuff

-- time passing
channel tock

-- ages of pieces of information
maxAge = 1
Age = {0..maxAge}
nextAge(t) = if t==maxAge then maxAge else t+1

-- time after which network is expected to have stabilised
stabilityTime = 1 -- this seems to work

---- functions over sequences

-- test whether ns starts with <i,j>

startsWith(i,j,<i',j'>^ns) = i==i' and j==j'

second(<_,n>^ns) = n

last(<n>) = n
last(<n>^ns) = last(ns)

----------------------------------------------------------------------
---------- Messages and beliefs
{-
We'll represent messages going round the network by

  Node . Node . Status . Age

tuples.  Each i.j.s.t represents that i said t time units ago that the link
to j had status s.  All such messages will have i!=j.

Each node will maintain a set "beliefs" of messages that it has received and
that have not been superceded, representing its beliefs about the state of the
```

```
network.
-}

-- sending and signalling of beliefs
channel send : Node . Node . Node . Node . Status . Age
channel signal : Node . LinkSet . Status

----------------------------------------------------------------------
---------- The link processes

-- Channels signal breaking or being made using the following channels

channel make, break : LinkSet

-- A node is either made or broken

Link(l) = Made(l)

Made(l) =
  send ? i:l ? j:diff(l,{i}) ? m -> Made(l)
  []
  break.l -> Broken(l)

Broken(l) = make.l -> Made(l)

Links = ||| l : LinkSet @ Link(l)

----------------------------------------------------------------------
---------- Setting providers
{-
We will use the following local events within each node for setting the
provider:

* setProviderInt . <id,p,j_1,...,j_k,i> represents that id sets the
  provider for i to be p (where maybe i=p) because of a believed path
  id -> p -> j_1 ->...-> j_k -> i.

* setProvider . id . i . p represents that the local node id sets the
  provider for i to be p.

* disconnected .id . ns represents that the nodes in ns are disconnected
  from the local node id.
-}

-- All sequences of distinct nodes of length at least 2

AllNodeSeqs =
  let -- all sequences of elements from given sequence
      allNodeSeqs'(<>) = {<>}
      allNodeSeqs'(<n>^ns) =
        let ans = allNodeSeqs'(ns)
        within union( ans, Union({ insert(n, ns') | ns' <- ans }) )
      -- all ways of inserting n into given sequence
      insert (n,<>) = {<n>}
      insert (n,<n'>^ns) =
        union( {<n,n'>^ns}, {<n'>^ns' | ns' <- insert(n,ns)} )
  within { ns | ns <- allNodeSeqs'(<1..N>), length(ns)>=2 }
```

```
channel setProvider : Node . Node . Node
channel setProviderInt : AllNodeSeqs
channel disconnected : Node . Set(Node)

----------------------------------------------------------------------
---------- The node processes
{-
We will represent each node by a parallel composition:

* one process for each link in the graph;

* one process to keep track of the provider for each other node.

Throughout the following, id will be the identity of the local node.
-}

-- Keep track of state of local link from id to j; s is current status

Local(id, j, s) =
  ( -- detect link going up or down
    if s==Down then make.{id,j} -> Local(id, j, Up)
    else break.{id,j} -> Local(id, j, Down)
  )
  [] -- send belief to other node (i), with age 0
  send.id?i:other(id)!id.j.s.0 -> Local(id, j, s)
  [] -- signal belief
  signal.id.{id,j}.s -> Local(id, j, s) -- don't really need this
  [] -- allow provider to be set
  (
    if s==Up then
      [] ns : AllNodeSeqs, startsWith(id,j,ns) @
        setProviderInt.ns -> Local(id, j, s)
    else
      [] ns : Set(diff(Node, {id,j})) @
        disconnected.id.(union(ns,{j})) -> Local(id, j, s)
  )

-- alphabet of Local(id, j, _)

aLocal(id,j) =
  Union({
    { make.{id,j}, break.{id,j} },
    {| signal.id.{id,j} |},
    {| send.id.i.id.j | i <- other(id) |},
    { setProviderInt.ns | ns <- AllNodeSeqs, startsWith(id,j,ns) },
    { disconnected.id.(union(ns,{j})) | ns : Set(diff(Node, {id,j})) }
  })

-- Keep track of status of link {i,j}, not local to id.
-- Node i was last of i and j from whom we accepted a message.
-- i and j can swap during execution.
-- Status is s.  Information has age t.

Remote(id, i, j, s, t) =
  -- send information to other node (k)
  send.id?k:other(id)!i.j.s.t -> Remote(id, i, j, s, t)
```

```
  []
  ( -- receive information from other node (k);
    -- will be accepted only if k is provider for i'
    [] k:other(id), i':{i,j}, j':diff({i,j},{i'}) @
      send.k.id.i'.j'?s'?t':{0..t-1} -> Remote(id, i', j', s', t')
  )
  [] -- allow time to pass
  tock -> Remote(id, i, j, s, nextAge(t))
  [] -- signal belief
  signal.id.{i,j}.s -> Remote(id, i, j, s, t)
  [] -- allow provider to be set, or not if it's disconnected
  (
    if s==Up then
      [] ns : AllNodeSeqs, subseq(i,j,ns) or subseq(j,i,ns) @
        setProviderInt.ns -> Remote(id, i, j, s, t)
    else
      [] ns : Set(diff(Node, {id,i,j})) @
        disconnected.id.(union(ns,{j})) -> Remote(id, i, j, s, t)
        []
        disconnected.id.(union(ns,{i})) -> Remote(id, i, j, s, t)
  )
  []
  ( -- identify that i and j are disconnected
    [] ns : Set(diff(Node, {id,i,j})) @
      disconnected.id.(union(ns,{i,j})) -> Remote(id, i, j, s, t)
  )

-- test whether <i,j> is a subsequence of ns

subseq(i,j,<_>) = false
subseq(i,j,ns) = startsWith(i,j,ns) or subseq(i,j,tail(ns))

-- alphabet of Remote(id, i, j, _, _) and Remote(id, j, i, _, _)

aRemote(id, i, j) =
  Union({
    {| send.id.k.i.j, send.k.id.i.j, send.id.k.j.i, send.k.id.j.i |
         k <- other(id) |},
    { tock },
    {| signal.id.{i,j} |},
    { setProviderInt.ns |
        ns <- AllNodeSeqs, subseq(i,j,ns) or subseq(j,i,ns) },
    { disconnected.id.union(ns,ns') |
        ns <- Set(diff(Node, {id,i,j})), ns' <- {{i}, {j}, {i,j}} }
  })

-- Combine above processes in parallel

Arcs(id) =
  let nodePairs =
        union(
          { (id,j) | j <- other(id) },
          { (i,j) | i <- other(id), j <- {i+1..N}, j!=id }
        )
  within
    || (i,j): nodePairs @
      [if i==id then aLocal(id, j) else aRemote(id, i, j)]
```

```
      if i==id then Local(id, j, Up) else Remote(id, i, j, Up, 0)

-- Now build processes to keep track of providers to id.

-- First, a process to keep track of the provider for i to id.  p represents
-- the current provider if id believes it is connected to i; otherwise p is
-- set to 0.

ChooseProvider(id, i, p) =
  ( -- set the provider for i to be p
    [] ns : AllNodeSeqs, head(ns)==id, last(ns)==i @
      setProviderInt.ns -> ChooseProvider(id, i, second(ns))
  )
  []
  ( -- discover that we're disconnected
    [] ns : Set(diff(Node, {id,i})) @
      disconnected.id.union(ns,{i}) -> ChooseProvider(id, i, 0)
  )
  [] -- allow send of message originating at i from p to here
  p!=0 & send.p.id.i?j?s?t -> ChooseProvider(id, i, p)

aChooseProvider(id, i) =
  Union({
    { setProviderInt.ns | ns <- AllNodeSeqs, head(ns)==id, last(ns)==i },
    {| send.p.id.i | p <- other(id) |},
    { disconnected.id.union(ns,{i}) | ns <- Set(diff(Node, {id,i})) }
  })

ChooseProviders(id) =
  || i:other(id) @ [aChooseProvider(id, i)] ChooseProvider(id, i, i)

-- Buld agent from components

Agent00(id) =
  let SyncSet =
        union(
          {| setProviderInt, disconnected.id |},
          {| send.p.id | p <- other(id) |}
        )
  within Arcs(id) [| SyncSet |] ChooseProviders(id)

Agent0(id) =
  if N<=2 then   -- otherwise we have a time-stop!
    let TS = tock -> TS within Agent00(id) ||| TS
  else Agent00(id)

Agent(id) =
    Agent0(id)
      [[ setProviderInt.ns <- setProvider.id.last(ns).second(ns) |
           ns <- AllNodeSeqs, head(ns)==id
      ]]

-- Alphabet of agent id

alpha(id) =
  Union({
    { make.{id,i}, break.{id,i} | i <- other(id) },
```

```
    {| send.id.i, send.i.id | i <- other(id) |},
    { tock },
    {| signal.id, setProvider.id |},
    { disconnected.id.ns | ns <- Set(diff(Node, {id})) }
  })

Agents = || id : Node @ [alpha(id)] Agent(id)

----------------------------------------------------------------------

{-
We add a global scheduling agent, in order to reduce the state space.  The
scheduler forces agents to choose their providers in a particular order:

* Initially, force all agents to choose their providers in order

* Force each agent i to re-choose its providers after receiving a message;

* Force agents i and j to re-choose providers after a change in the local link
  {i,j}

* Allow other events only once all providers have been so re-chosen.
-}

Scheduler = Scheduler0(<1..N>,1)

-- Force agents in <i>^is to choose providers, in order; i should choose
-- provider for j next

Scheduler0(<i>^is, j) =
  j<=N and j!=i & (
    setProvider.i.j?p -> Scheduler0(<i>^is, j+1)
    []
    [] ns : Set(diff(Node, {i,j})) @
      disconnected.i.(union(ns,{j})) -> Scheduler0(<i>^is, j+1)
  )
  []
  j==i & Scheduler0(<i>^is, j+1)
  []
  j>N & Scheduler0(is, 1)

-- allow other events only in this state

Scheduler0(<>, j) = Scheduler1

Scheduler1 =
  let order(l) = -- turn link into ordered sequence of nodes
        let {is} = {<i,j> | i <- l, j <- l, i<j} within is
  within
  (
    [] l : LinkSet @
      make.l -> Scheduler0(order(l),1) [] break.l -> Scheduler0(order(l),1)
  ) -- force nodes local to l to choose scheduler
  []
  send?i?j?x -> Scheduler0(<j>,1) -- force j to choose scheduler
  []
  tock -> Scheduler1
```

```
  []
  signal?i?x -> Scheduler1

---------------------------------------------------------------------
---------- The system

System00 =
  let syncSet = {| make, break, send, tock, signal, disconnected, setProvider|}
  within
  ( Agents [| syncSet |] Scheduler )

System0 = Links [| {| make, break, send |} |] System00

System = System0 \ {| send, disconnected, setProvider |}

---------------------------------------------------------------------
---------- The specification

{-
The specification keeps track of the connectivity of the network in the
parameter graph (a set of edges).  If node id is connected to the link {i,j}
then when the network settles it should report the truth about that link.  The
settling will occur within at most stabilityTime (= 1) tocks of the topology
changing; instability records how soon it should stabilise.
-}

Spec(graph, instability) =
  let connectedToLink(i, l) = -- is node i connected to link l?
        inter( l, connectedCmpt(i) ) != {}
      connectedCmpt(i) = -- calculate connected component of i
        step({i}, other(i))
      step(connected, unknown) =
        -- iterative calculation of connected component; connected are those
        -- nodes that we know to be in this component; unknown is the rest
        let newConnected =
              { j | k <- connected, j <- unknown, member({k,j}, graph) }
        within
          if newConnected=={} then connected
          else
            let connected' = union(connected, newConnected)
                unknown' = diff(unknown, newConnected)
            within step(connected', unknown')
  within
  -- allow links to make or break
  make?l:diff(LinkSet,graph) -> Spec(union(graph, {l}), stabilityTime)
  []
  break?l:graph -> Spec(diff(graph, {l}), stabilityTime)
  []
  (
    if instability>0 then
      -- allow any signals, but do not insist they are available
      signal?id?l?s -> Spec(graph, instability) |~| STOP
    else (
      -- allow any signals initially
      signal?id?l?s -> Spec(graph, instability)
      [> ( -- but settle into state where only correct signals are available
        [] id : Node, l : LinkSet @
```

```
          if connectedToLink(id, l) then
            signal.id.l.(if member(l,graph) then Up else Down) ->
            Spec(graph, instability)
          else
            |~| s : Status @ signal.id.l.s -> Spec(graph, instability)
      )
    )
  )
  [] -- allow time to pass
  tock -> Spec(graph, if instability>0 then instability-1 else 0)

-- Test whether the specification holds

assert Spec(LinkSet,0) [FD= System

{-

This doesn't hold, because after the following trace

setProvider.1.2.2
setProvider.1.3.2
setProvider.2.1.1
setProvider.2.3.1
setProvider.3.1.2  * 3 decides 2 is the provider for 1
setProvider.3.2.1  * 3 decides 1 is the provider for 2
break.{1,2}        * link {1,2} goes down
setProvider.1.2.3
setProvider.1.3.3
setProvider.2.1.3
setProvider.2.3.3
tock
send.3.2.3.1.Up.0
setProvider.2.1.3
setProvider.2.3.3
send.3.1.3.2.Up.0
setProvider.1.2.3
setProvider.1.3.3

the system refuses

signal.3.{1,2}.Down

because 3 can never hear about the link going down.
-}
```

B Second CSP model

```
-- Reverse Path Forwarding, combined with routing information

N = 4 -- number of nodes

----- Type of nodes:

Node = {1..N}
other(i) = diff(Node, {i})

----- Type of links; uncomment the one you want to try

-- LinkSet = { {i,j} | i <- Node, j <- other(i) }  -- complete graph
LinkSet = { {i,j} | i <- Node, j <- other(i), i%N+1==j } -- ring
-- LinkSet = { {i,j} | i <- Node, j <- other(i), i+1==j } -- linear network

----- status of channels

datatype Status = Up | Down

otherStatus(s) = if s==Up then Down else Up

-- timing stuff

-- time passing
channel tock

-- ages of pieces of information
maxAge = 1
Age = {0..maxAge}
nextAge(t) = if t==maxAge then maxAge else t+1

-- time after which network is expected to have stabilised
stabilityTime = 1 -- this seems to work

-----------------------------------------------------------------------
---------- Messages and beliefs
{-
We'll represent messages going round the network by

  Node . Node . Status . Age

tuples.  Each i.j.s.t represents that i said t time units ago that the link
to j had status s.  All such messages will have i!=j.

Each node will maintain a set "beliefs" of messages that it has received and
that have not been superceded, representing its beliefs about the state of the
network.
-}

-- sending and signalling of beliefs
channel send : Node . Node . Node . Node . Status . Age
channel signal : Node . LinkSet . Status

-----------------------------------------------------------------------
```

```
-- making and breaking of nodes:
channel make, break : LinkSet

----------------------------------------------------------------------
---------- Setting providers
{-
We will use the following local events within each node for setting the
provider:

* setProviderInt . tree represents that the local node sets the providers to
  be as specified by the tree tree.  This is renamed to setProviders . id

* setProviders . id represents that the local node id sets its providers.
-}

-- calculate connected component of i in graph

connectedCmpt(i, graph) =
  let step(connected, unknown) =
        let newConnected =
              { j | k <- connected, j <- unknown, member({k,j}, graph) }
        within
          if newConnected=={} then connected
          else
            let connected' = union(connected, newConnected)
                unknown' = diff(unknown, newConnected)
            within step(connected', unknown')
  within step({i}, other(i))

-- calculate all trees rooted at id

Trees(id) =
  { ls | ls <- Set(LinkSet), card(connectedCmpt(id, ls)) == card(ls)+1}

AllTrees = Union({ Trees(id) | id <- Node })

-- nodes linked by tree

span(tree) = Union(tree)

-- declare channels

channel setProviderInt : AllTrees

channel setProviders : Node

----------------------------------------------------------------------
-- The node processes

-- Represent Agent by parallel composition:

-- * one process for each link in the graph;

-- * one process to keep track of the provider for each other node.

-- Keep track of state of local link from id to j; s is current status
```

```
Local(id, j, s) =
  ( -- detect link going up or down
    if s==Down then make.{id,j} -> Local(id, j, Up)
    else break.{id,j} -> Local(id, j, Down)
  )
  [] -- send belief to other node (i), with age 0; no point sending to j
  send.id?i:diff(Node,{id,j})!id.j.s.0 -> Local(id, j, s)
  [] -- allow send on this link if up
  s==Up & (
    [] i : Node, k : other(i), {i,k}!={id,j}, member({i,k},LinkSet) @
      send.id.j.i.k?s' -> Local(id, j, s)
      []
      send.j.id.i.k?s' -> Local(id, j, s)
  )
  [] -- signal belief
  signal.id.{id,j}.s -> Local(id, j, s) -- don't really need this
  [] -- allow provider to be set
  (
    if s==Up then
      [] tree : Trees(id), member({id,j}, tree) @
        setProviderInt.tree -> Local(id, j, s)
    else
      [] tree : Trees(id), not(member(j, span(tree))) @
        setProviderInt.tree -> Local(id, j, s)
  )

-- alphabet of Local(id, j, _)

aLocal(id,j) =
  Union({
    { make.{id,j}, break.{id,j} },
    {| signal.id.{id,j} |},
    {| send.id.i.id.j, send.id.i.j.id | i <- diff(Node,{id,j}) |},
    {| send.id.j, send.j.id |},
    { setProviderInt.tree |
        tree <- Trees(id),
        member({id,j}, tree) or not(member(j, span(tree))) }
  })

-- Status of link {i,j}, not local to id.  i was last one to report to us;
-- i and j can swap during execution.  Status is s.  Information has age t.

Remote(id, i, j, s, t) =
  -- send information to other node (k); no point telling i or j
  send.id?k:diff(Node,{id,i,j})!i.j.s.t -> Remote(id, i, j, s, t)
  []
  ( -- receive information from other node (k);
    -- will be accepted only if k is provider for i'
    [] k:other(id), i':{i,j}, j':diff({i,j},{i'}) @
      send.k.id.i'.j'?s'?t':{0..t-1} -> Remote(id, i', j', s', t')
  )
  [] -- allow time to pass
  tock -> Remote(id, i, j, s, nextAge(t))
  [] -- signal belief
  signal.id.{i,j}.s -> Remote(id, i, j, s, t)
  [] -- allow providers to be set
  (
```

```
    if s==Up then
      [] tree : Trees(id), member({i,j}, tree) @
        setProviderInt.tree -> Remote(id, i, j, s, t)
    else
      [] tree : Trees(id),
         -- test whether {i,j} links the tree to the other nodes
         card( inter( {i,j}, span(tree) ) ) == 1 @
        setProviderInt.tree -> Remote(id, i, j, s, t)
  )

-- alphabet of Remote(id, i, j, _, _) and Remote(id, j, i, _, _)

aRemote(id, i, j) =
  Union({
    {| send.id.k.i.j, send.k.id.i.j, send.id.k.j.i, send.k.id.j.i |
         k <- other(id) |},
    { tock },
    {| signal.id.{i,j} |},
    { setProviderInt.tree |
        tree <- Trees(id),
        member({i,j}, tree) or card( inter( {i,j}, span(tree) ) ) == 1 }
  })

-- state of all arcs as believed by node id

Arcs(id) =
  let nodePairs =
        union(
          { (id,j) | j <- other(id), member({id,j}, LinkSet) },
          { (i,j) | i <- other(id), j <- {i+1..N},
                    j!=id, member({i,j}, LinkSet) }
        )
  within
    || (i,j): nodePairs @
      [if i==id then aLocal(id, j) else aRemote(id, i, j)]
       if i==id then Local(id, j, Up) else Remote(id, i, j, Up, 0)

-- keep track of provider for i to id; current provider is p

ChooseProvider(id, i, p) =
  ( -- set the provider for i to be p
    [] tree : Trees(id) @
      setProviderInt.tree -> ChooseProvider(id, i, providerFor(id, i, tree))
  )
  [] -- allow send of message originating at i from p to here
  p!=0 & send.p.id.i?j?s?t -> ChooseProvider(id, i, p)

-- Find out who is provider for i to id in tree:; if i is connected to id,
-- then the provider is the j such that {id,j} is in the tree, but i is not
-- connected to id when {id,j} is removed.

providerFor(id, i, tree) =
  if member(i, span(tree)) then
    let { j } = { j | j <- other(id), member({id,j}, tree),
                      not(member(i, connectedCmpt(id, diff(tree,{{id,j}})))) }
    within j
  else
```

```
    0

aChooseProvider(id, i) =
  Union({
    { setProviderInt.tree | tree <- Trees(id) },
    {| send.p.id.i | p <- other(id) |}
  })

ChooseProviders(id) =
  || i:other(id) @ [aChooseProvider(id, i)] ChooseProvider(id, i, i)

-- Buld agent from components

Agent00(id) =
  let SyncSet = union( {| setProviderInt |}, {| send.p.id | p <- other(id) |} )
  within Arcs(id) [| SyncSet |] ChooseProviders(id)

Agent0(id) =
  if N<=2 then   -- otherwise we have a time-stop!
    let TS = tock -> TS within Agent00(id) ||| TS
  else Agent00(id)

Agent(id) =
  Agent0(id) [[ setProviderInt.tree <- setProviders.id | tree <- Trees(id) ]]

-- Alphabet of agent id

alpha(id) =
  Union({
    { make.{id,i}, break.{id,i} | i <- other(id), member({id,i}, LinkSet) },
    {| send.id.i, send.i.id | i <- Node |},
    { tock },
    {| signal.id, setProviders.id |}
  })

----------------------------------------------------------------------

{-
We add a global scheduling agent, in order to reduce the state space.  The
scheduler forces agents to choose their providers in a particular order:

* Initially, force all agents to choose their providers in order

* Force each agent i to re-choose its providers after receiving a message;

* Force agents i and j to re-choose providers after a change in the local link
  {i,j}

* Allow other events only once all providers have been so re-chosen.
-}
Scheduler = Scheduler0(<1..N>)

-- Force agents in <i>^is to choose providers, in order

Scheduler0(<i>^is) = setProviders.i -> Scheduler0(is)

-- allow other events only in this state
```

```
Scheduler0(<>) = Scheduler1

Scheduler1 =
  let order(l) = -- turn link into ordered sequence of nodes
        let {is} = {<i,j> | i <- l, j <- l, i<j} within is
  within
  (
    [] l : LinkSet @
      make.l -> Scheduler0(order(l)) [] break.l -> Scheduler0(order(l))
  ) -- force nodes local to l to choose scheduler
  []
  send?i?j?x -> Scheduler0(<j>) -- force j to choose scheduler
  []
  tock -> Scheduler1
  []
  signal?i?x -> Scheduler1

----------------------------------------------------------------------
-- The system

transparent sbisim
transparent diamond
compress1(P) = sbisim(diamond(P))

Agents = || id : Node @ [alpha(id)] compress1(Agent(id))

System =
  let syncSet = {| make, break, send, tock, signal, setProviders|}
  within ( Agents [| syncSet |] Scheduler ) \ {| setProviders, send |}

----------------------------------------------------------------------
---------- The specification

{-
The specification keeps track of the connectivity of the network in the
parameter graph (a set of edges).  If node id is connected to the link {i,j}
then when the network settles it should report the truth about that link.  The
settling will occur within at most stabilityTime tocks of the topology
changing; instability records how soon it should stabilise.
-}

channel anySignal


Spec0(graph, instability) =
  let connectedToLink(i, l) =  -- is node i connected to link l?
        inter( l, connectedCmpt(i, graph) ) != {}
  within
  -- allow links to make or break
  make?l:diff(LinkSet,graph) -> Spec0(union(graph, {l}), stabilityTime)
  []
  break?l:graph -> Spec0(diff(graph, {l}), stabilityTime)
  []
  (
    if instability>0 then
      -- allow any signals, but do not insist they are available
```

```
      anySignal -> Spec0(graph, instability) [> STOP
    else (
      -- allow any signals initially
      anySignal -> Spec0(graph, instability)
      [> ( -- but settle into state where only correct signals are available
        [] id : Node, l : LinkSet @
          if connectedToLink(id, l) then
            signal.id.l.(if member(l,graph) then Up else Down) ->
            Spec0(graph, instability)
          else
            STOP -- |~| s : Status @ signal.id.l.s -> Spec0(graph, instability)
      )
    )
  )
  [] -- allow time to pass
  tock -> Spec0(graph, if instability>0 then instability-1 else 0)

Spec =
  Spec0(LinkSet,0)
    [[ anySignal <- signal.id.l.s | id <- Node, l <- LinkSet, s <- Status ]]

-- Test whether the specification holds

assert Spec [F= System
```

C First Alloy model

```
module rpf

// In this document the measure to determine the shortest path is the
// number of hops.

// This will be the same as newRPF6.als except that a node can update
// some or all of its information.

// It works for a four-node chain.
// It works for a four-node ring.

--                      TYPE DECLARATIONS

sig Configuration {}
static part sig Ring, Chain, CompleteGraph extends Configuration {}

sig NodeId {}

sig Trace {}
disj sig EmptyTrace extends Trace {}
disj sig NonEmptyTrace extends Trace {
  last : Op,
  prefix : Trace
}

sig Op { link : Link }
disj sig Make, Break, Propagate extends Op {}

sig Distance {}
static part sig One, Two, Three, Four extends Distance {}

sig Age {}
static part sig Recent, Old extends Age {}

--                      DATA TYPES

//  A network records the configuration of the system and its
//  maximalGraph (a set containing all possible links) as well as
//  all links that are up.  In addition it records the current state
//  of each node.

sig Network {
  config : Configuration,
  maximalGraph : set Link,
  upLinks : set Link,
  nodes : set Node,
  trace : Trace
}

// A link has a pair of NodeIds labelling its ends.
sig Link {
  ends : set NodeId
} {
  # ends = 2
}
```

```
// A timed link also records when that the status of the link was recorded.
sig TimedLink {
  link : Link,
  age : Age
}

//  Each node records its own id and its belief about the status of
//  each link.
sig Node {
  id : NodeId,
  believesUp, believesDown : set TimedLink,
  -- The nodes (excluding itself) it believes it is connected to.
  believesConnectedTo : set NodeId,
  -- The following states that onPathTo is a (partial) function.
  -- It relates states to the first node on their shortest path.
  distance : NodeId ->? Distance,
  onPathTo : NodeId ->? NodeId
}

--                      CANONICALISATION

fact {
  all n, n' : Network {
    ( n.config = n'.config && n.maximalGraph = n'.maximalGraph &&
      n.upLinks = n'.upLinks && n.nodes = n'.nodes &&
      n.trace = n'.trace )
         => n = n'
  }
  all l, l' : Link {
    l.ends = l'.ends => l = l'
  }
  all l,l' : TimedLink {
    ( l.link = l'.link && l.age = l'.age ) => l = l'
  }
  all n, n' : Node {
    ( n.id = n'.id && n.believesUp = n'.believesUp &&
      n.believesDown = n'.believesDown &&
      n.onPathTo = n'.onPathTo
      // The remaining components will be equivalent by construction.
    )
         => n = n'
  }
  all w, w' : WorldView {
    ( w.initial = w'.initial && w.final = w'.final )
        => w = w'
  }
}

--                      CONFIGURATION

//  The maximalGraph of a network reflects the configuration.
fact {
  all network : Network {
    ( network.config = CompleteGraph
        =>
          all i, j : NodeId {
```

```
          i != j
            =>
              some link : network.maximalGraph {
                (i + j) = link.ends
              }
         }
    )
    &&
    ( // This holds for up to (and including) 5 nodes.
      network.config = Ring
        =>
        all i  : NodeId {
          # { link : network.maximalGraph | i in link.ends } = 2
        }
    )
    &&
    ( // This holds for up to (but EXcluding) 5 nodes.
      network.config = Chain
        =>
          ( # { i : NodeId |
                # { link : network.maximalGraph | i in link.ends } = 1 }
                   = 2
           &&
           2 + # { i : NodeId |
                   # { link : network.maximalGraph
                                        | i in link.ends } = 2 }
                      = # NodeId
          )
    )
  }
}

--                      IMPLIED STATE COMPONENTS

// Some components of the state may be derived / infered from others
// but are recorded for convenience.  These include the
// believesConnectedTo, distance and onPathTo components of a node.

// The following definition for believesConnectedTo holds for
// up to (and including) 5 nodes.
fact {
  all node : Node {
    node.believesConnectedTo =
      { j : NodeId | j != node.id &&
         some link, link', link'', link''' : node.believesUp.link {
           (node.id + j) in
                (link.ends + link'.ends + link''.ends + link'''.ends)
           &&
           some ( link.ends & link'.ends )
           &&
           some ( link'.ends & link''.ends )
           &&
           some ( link''.ends & link'''.ends )
         }
      }
  }
}
```

```
// The set believesConnectedTo recorded by a node is the domain of the
// functions distance and onPathTo.
fact {
  all node : Node {
    (node.distance).Distance = node.believesConnectedTo
    (node.onPathTo).NodeId = node.believesConnectedTo
  }
}

// The distance function (maximum value Four for up to five nodes) is
// constructed as follows.
fact {
  all node : Node {
    all dest : node.believesConnectedTo {

      // A node is distance One precisely when there is a direct link to it.
      dest in (node.distance).One <=>
        some link : node.believesUp.link { (dest + node.id) in link.ends }

      // A node is of distance Two precisely when it is not itself a node
      // of distance One but is connected to a node of distance One.
      ( some link, link' : node.believesUp.link {
          ( node.id in link.ends ) && ( dest in link'.ends ) &&
          ( some (link.ends & link'.ends) )
        }
       )  => dest in ( (node.distance).One + (node.distance).Two )

      // A node is of distance Three precisely when it is not itself a node
      // of distance One or Two but is connected to a node of distance Two.
      ( some link, link',link'' : node.believesUp.link {
          node.id in link.ends && dest in link''.ends &&
          ( some (link.ends & link'.ends) ) &&
          ( some (link'.ends & link''.ends) )
        }
      )   => dest in ( (node.distance).One + (node.distance).Two +
                       (node.distance).Three )

      // A node is of distance Four precisely when it is not itself a node
      // of distance One, Two or Three but is connected to a node of
      // distance Three.
      ( some link, link',link'',link''' : node.believesUp.link {
          node.id in link.ends && dest in link'''.ends &&
          ( some (link.ends & link'.ends) ) &&
          ( some (link.ends & link''.ends) ) &&
          ( some (link'.ends & link'''.ends) )
        }
      )   => dest in ( (node.distance).One + (node.distance).Two +
                       (node.distance).Three + (node.distance).Four )

    }
  }
}

// The onPathTo function is constructed as follows
fact {
  all node : Node {
```

```
    // A node can only be the first node on the path to another node
    // if it is of distance One away.
    NodeId.(node.onPathTo) in (node.distance).One

    // A node of distance One away is the first node on the path to itself.
    all dest : (node.distance).One { dest.(node.onPathTo) = dest }

    // The first node on the path to a node of distance Two must be
    // directly connected to that node.
    all dest : (node.distance).Two {
      some link : node.believesUp.link {
        (link.ends - dest) in dest.(node.onPathTo)
      }
    }

    // The first node on the path to a node of distance Three or Four
    // must be indirectly connected to that node.
    all dest : (node.distance).Three {
      some link, link' : node.believesUp.link {
        ( some ( link.ends & link'.ends ) ) &&
        (link.ends - link'.ends) in dest.(node.onPathTo) &&
        (link'.ends - link.ends) = dest
      }
    }
    all dest : (node.distance).Four {
      some link, link', link'' : node.believesUp.link {
        ( some ( link.ends & link'.ends ) ) &&
        ( some ( link'.ends & link''.ends) ) &&
        (link.ends - link'.ends) in dest.(node.onPathTo) &&
        (link'.ends - link.ends) in link''.ends
        (link''.ends - link'.ends) = dest
      }
    }
  }
}

--                      FACTS

// Some facts about the beliefs of nodes.
fact {
  // Every node has a single belief about every link.
  all net : Network {
    all n : net.nodes {
      n.believesUp.link + n.believesDown.link = net.maximalGraph &&
      no (n.believesUp.link & n.believesDown.link)
    }
  }
  // No node has both old and new opinions about the status of a link.
  all n : Node {
    all timedLink, timedLink' : ( n.believesUp + n.believesDown ) {
      timedLink.link = timedLink'.link => timedLink = timedLink'
    }
  }
  // Knowledge of the status of adjacent links is always recent.
  all n : Node {
    all timedLink : (n.believesUp + n.believesDown) {
```

```
      n.id in timedLink.link.ends => timedLink.age in Recent
    }
  }
}

// Some facts about the network
fact {
  // No two nodes in a network have the same id
  all net : Network {
    all n, n' : net.nodes {
      n.id = n'.id => n = n'
    }
  }
  // Every node is included in the maximal graph of a network.
  all net : Network {
    net.nodes.id = NodeId
  }
}

--                      INITIAL STATE

// Initially all links are up and all nodes believe that all links
// are up.

fun InitialisedNetwork (network : Network) {
  network.upLinks = network.maximalGraph
  all n : network.nodes {
    InitialisedNode (n)
  }
  network.trace in EmptyTrace
}

fun InitialisedNode (node : Node) {
  no node.believesDown
  node.believesUp.age in Recent
}

--                      WORLD VIEW

// The WorldView records two networks: the initial network and the
// final one after some sequence of operations, as well as the trace that
// led from the first network to the second.
sig WorldView {
  initial, final : Network,
  makeLink, breakLink, propagate : Network -> Network
} {
  InitialisedNetwork(initial)
  all net, net' : Network {
    ( net' in makeLink[net] <=> MakeLink(net,net'))
    &&
    ( net' in breakLink[net] <=> BreakLink(net,net'))
    &&
    ( net' in propagate[net] <=> PropagateInfo(net,net'))

  }
  final in (^(makeLink + breakLink + propagate))[initial]
}
```

```
--                    FUNCTIONS OF THE SYSTEM

fun BreakLink (net,net' : Network) {
  // one link has been broken and the others remain unchanged.
  net'.upLinks in net.upLinks
  # net'.upLinks + 1 = # net.upLinks
  // the configuration and the maximal graph remain unchanged.
  net.config = net'.config && net.maximalGraph = net'.maximalGraph
  // the Break operation is added to the trace
  net'.trace.last in Break && net'.trace.prefix = net.trace
  // Observe that there is precisely one link in (net.upLinks - net'.upLinks)
  all breakingLink : (net.upLinks - net'.upLinks) {
    // the trace records which link was broken.
    net'.trace.last.link = breakingLink
    &&
    // The state of the nodes is changed as follows:
    all oldNode : net.nodes {
      some newNode : net'.nodes {
        // There is a new node for every old node.
        newNode.id = oldNode.id &&
        // Nodes not connected to the breaking link keep the same beliefs.
        ( newNode.id !in breakingLink.ends =>
            newNode.believesUp.link = oldNode.believesUp.link ) &&
        // Nodes connected to the breaking link change only their
        // belief about that link.
        ( newNode.id in breakingLink.ends =>
            newNode.believesUp.link =
              ( oldNode.believesUp.link - breakingLink ) ) &&
        // All beliefs about indirect links are now old.
        all timedLink : newNode.(believesUp + believesDown) {
          newNode.id !in timedLink.link.ends => timedLink.age in Old
        }
      }
    }
  }
}

fun MakeLink (net,net' : Network) {
  // one link has been broken and the others remain unchanged.
  net.upLinks in net'.upLinks
  # net.upLinks + 1 = # net'.upLinks
  // the configuration and the maximal graph remain unchanged.
  net.config = net'.config && net.maximalGraph = net'.maximalGraph
  // the Make operation is added to the trace
  net'.trace.last in Make && net'.trace.prefix = net.trace
  // Observe that there is precisely one link in (net'.upLinks - net.upLinks)
  all makingLink : (net'.upLinks - net.upLinks) {
    // the trace records which link was broken.
    net'.trace.last.link = makingLink
    &&
    // The state of the nodes is changed as follows:
    all oldNode : net.nodes {
      some newNode : net'.nodes {
        // There is a new node for every old node.
        newNode.id = oldNode.id &&
        // Nodes not connected to the making link keep the same beliefs.
```

```
        ( newNode.id !in makingLink.ends =>
            newNode.believesUp.link = oldNode.believesUp.link ) &&
        // Nodes connected to the breaking link change only their
        // belief about that link.
        ( newNode.id in makingLink.ends =>
            newNode.believesUp.link =
              ( oldNode.believesUp.link + makingLink ) ) &&
        // All beliefs about indirect links are now old.
        all timedLink : newNode.(believesUp + believesDown) {
          newNode.id !in timedLink.link.ends => timedLink.age in Old
        }
      }
    }
  }
}

// We now define a function that is true precisely when there is some
// information to be propagated.

fun NotOmniscient (net : Network) {
  some seeksInfo, hasInfo : net.nodes {
    // oldLink is a link of seeksInfo and recentLink is a link of hasInfo
    some oldLink : (seeksInfo.believesUp + seeksInfo.believesDown),
         recentLink : (hasInfo.believesUp + hasInfo.believesDown) {
      // they are talking about the same link...
      oldLink.link = recentLink.link &&
      /// but hasInfo has more recent knowledge
      oldLink.age in Old && recentLink.age in Recent &&
      // and hasInfo is first on the shortest path to one of the line ends.
      hasInfo.id in (seeksInfo.onPathTo)[oldLink.link.ends]
    }
  }
}

fun PropagateInfo (net,net' : Network) {
  // There is some information to be propagated
  NotOmniscient(net)
  // No links are broken or made.
  net'.upLinks = net.upLinks
  // the configuration and the maximal graph remain unchanged.
  net.config = net'.config && net.maximalGraph = net'.maximalGraph
  // the Propagate operation is added to the trace
  net'.trace.last in Propagate && net'.trace.prefix = net.trace
  // Every node in the old system corresponds to a node in the new system.
  all oldNode : net.nodes {
    some newNode : net'.nodes {
      newNode.id = oldNode.id
    }
  }
  // The state of only one node is changed.
  one (net'.nodes - net.nodes)
  // The before state of the changed node is thisNode.
  // The after state of the changed node is thisNode'.
  all thisNode : (net.nodes - net'.nodes),
      thisNode' : (net'.nodes - net.nodes) {
    // They have the same node id.
    thisNode'.id = thisNode.id &&
```

```
    // thisNode learns only one piece of information:
    (
    //   - it could gain recent information about a link that is up
      ( # (thisNode'.believesUp - thisNode.believesUp) = 1 &&
        # (thisNode.believesUp - thisNode'.believesUp) = 1 &&
        thisNode'.believesDown = thisNode.believesDown
      ) ||
    //   - it could gain recent information about a link that is down
      ( # (thisNode'.believesDown - thisNode.believesDown) = 1 &&
        # (thisNode.believesDown - thisNode'.believesDown) = 1 &&
        thisNode'.believesUp = thisNode.believesUp
      ) ||
    //   - it could learn that a down link is actually up
      ( thisNode.believesUp in thisNode'.believesUp &&
        # (thisNode'.believesUp - thisNode.believesUp) = 1 &&
        thisNode'.believesDown in thisNode.believesDown &&
        # (thisNode.believesDown - thisNode'.believesDown) = 1
      ) ||
    //   - or it could learn that an up link is actually down
      ( thisNode.believesDown in thisNode'.believesDown &&
        # (thisNode'.believesDown - thisNode.believesDown) = 1 &&
        thisNode'.believesUp in thisNode.believesUp &&
        # (thisNode.believesUp - thisNode'.believesUp) = 1
      )
    ) &&
    // The newly learnt piece of information is recent
    all timedLink : ( (thisNode'.believesUp - thisNode.believesUp) +
                      (thisNode'.believesDown - thisNode.believesDown)) {
      timedLink.age in Recent
    }
    // The replaced piece of information must have been old.
    all timedLink : ( (thisNode.believesUp - thisNode'.believesUp) +
                      (thisNode.believesDown - thisNode'.believesDown)) {
      timedLink.age in Old
    }
    // A piece of information can be learnt only if a node on the
    // shortest path to the link had recent information about that link.
    all timedLink : (thisNode'.believesUp - thisNode.believesUp) {
      some neighbour : net.nodes {
        // the neighbour is on the shortest path to the link
        neighbour.id in (thisNode.onPathTo)[timedLink.link.ends] &&
        // this ensures that the information is recent.
        timedLink in neighbour.believesUp
      }
    }
    all timedLink : (thisNode'.believesDown - thisNode.believesDown) {
      some neighbour : net.nodes {
        // the neighbour is on the shortest path to the link
        neighbour.id in (thisNode.onPathTo)[timedLink.link.ends] &&
        // this ensures that the information is recent.
        timedLink in neighbour.believesDown
      }
    }
  }
}

--                      SEARCHING FOR INCONSISTENCIES
```

```
// We define a function Inconsistent that is true whenever two nodes
// in a network have conflicting opinions about the status of a link.

// The following works for up to a 4 node ring and a 4 node chain.
fun Inconsistent (net : Network) {
  some node, node' : net.nodes {
    // The nodes must be connected
    some link, link', link'' : net.upLinks {
      node.id in link.ends && node'.id in link''.ends &&
      some (link.ends & link'.ends) && some (link'.ends & link''.ends)
    }
    // They have different beliefs about the status of a link.
    some node.believesUp.link & node'.believesDown.link
  }
}

// We define a function Error that is true whenever a network is
// stable (ie there is no more information to be propagated) and there
// is an inconsistency.

fun Error (net : Network) {
  // There is no more information to be propagated
  ( ! NotOmniscient(net) )
  // But there is an inconsistency.
  Inconsistent(net)
}


// FUNCTIONS

fun FourNodeRingShowMeError (world : WorldView) {
  # NodeId = 4
  world.initial.config = Ring
  Error(world.final)
}

// VERIFICATIONS
run FourNodeRingShowMeError for 3
  but 1 WorldView, 4 Distance, 2 Age, 4 Network, 4 Trace,
      2 Op, 4 NodeId, 14 Node, 4 Link, 8 TimedLink
  // 3 Configuration; 4 Distance; 2 Age - (since these are fixed)
  // 1 WorldView, 4 Network, 4 Trace, 2 Op
  // 4 NodeId, 14 Nodes, 4 Links, 8 TimedLinks
```

D Second Alloy model

```
module rpf

// In this document I have changed the distance measure: the
// shortest path is determined by a tree-like structure.

// We search a four node ring to see if we can find Gavin's error.

// The Propagate operation:

// * When I first defined the Propagate operation, a single node
//   learnt *all* information that it could possibly learn at that time.
//   Obviously, this will remove some behaviours.

// * Next (in newRPF8.als) only one piece of information was passed for
//   each operation.  Whilst this will preserve all behaviours, it will
//   make the state space unnecessarily large.

// * In this document we propagate any non-empty subset of the
//   information that could be propagated.

// The difference between this document and newRPF9.als is that we
// hard code in the structure of the maximal graph.



--                      TYPE DECLARATIONS

sig Configuration {}
static part sig Ring, Chain, CompleteGraph extends Configuration {}

sig NodeId {}
disj sig A, B, C, D extends NodeId {}

sig Trace {}
disj sig EmptyTrace extends Trace {}
disj sig NonEmptyTrace extends Trace {
  last : Op,
  prefix : Trace
}

sig Op { link : Link }
disj sig Make, Break, Propagate extends Op {}

sig Distance {}
static part sig One, Two, Three, Four extends Distance {}

sig Age {}
static part sig Recent, Old extends Age {}

--                      DATA TYPES

//  A network records the configuration of the system and its
//  maximalGraph (a set containing all possible links) as well as
//  all links that are up.  In addition it records the current state
//  of each node.
```

```
sig Network {
  config : Configuration,
  maximalGraph : set Link,
  upLinks : set Link,
  nodes : set Node,
  trace : Trace
}

// A link has a pair of NodeIds labelling its ends.
sig Link { ends : set NodeId } { # ends = 2 }

disj sig LinkAB extends Link {} { ends = A + B }
disj sig LinkBC extends Link {} { ends = B + C }
disj sig LinkCD extends Link {} { ends = C + D }
disj sig LinkDA extends Link {} { ends = D + A }

// A timed link also records when that the status of the link was recorded.
sig TimedLink {
  link : Link,
  age : Age
}

//  Each node records its own id and its belief about the status of
//  each link.
sig Node {
  id : NodeId,
  believesUp, believesDown : set TimedLink,
  -- The nodes (excluding itself) it believes it is connected to.
  believesConnectedTo : set NodeId,
  -- The following states that onPathTo is a (partial) function.
  -- It relates states to the first node on their shortest path.
  distance : NodeId ->? Distance,
  onPathTo : NodeId ->? NodeId
}

disj sig NodeA extends Node {} { id = A }
disj sig NodeB extends Node {} { id = B }
disj sig NodeC extends Node {} { id = C }
disj sig NodeD extends Node {} { id = D }


--                      CANONICALISATION

fact {
  all n, n' : Network {
    ( n.config = n'.config && n.maximalGraph = n'.maximalGraph &&
      n.upLinks = n'.upLinks && n.nodes = n'.nodes &&
      n.trace = n'.trace )
         => n = n'
  }
  all l, l' : Link {
    l.ends = l'.ends => l = l'
  }
  all l,l' : TimedLink {
    ( l.link = l'.link && l.age = l'.age ) => l = l'
  }
```

```
  all n, n' : Node {
    ( n.id = n'.id && n.believesUp = n'.believesUp &&
      n.believesDown = n'.believesDown &&
      n.onPathTo = n'.onPathTo
      // The remaining components will be equivalent by construction.
    )
         => n = n'
  }
  all w, w' : WorldView {
    ( w.initial = w'.initial && w.final = w'.final )
        => w = w'
  }
  all t, t' : Trace - EmptyTrace {
    ( t.last = t'.last && t.prefix = t'.prefix) => t = t'
  }
  all op, op' :  Make {
    ( op.link = op'.link ) => op = op'
  }
  all op, op' :  Break {
    ( op.link = op'.link ) => op = op'
  }
}

// Traces don't contain themselves.
fact {
  all t : Trace | t !in t.^prefix
}

// Every trace is either empty or non-empty.
// Every Op is either a send, a make or a break.
// Every Configuration is either a chain, a complete graph or a ring.
// Every Age is either old or recent
// Every distance is either one, two, three or four
fact {
   Trace in (EmptyTrace + NonEmptyTrace)
   Op in (Make + Break + Propagate)
   Configuration in (Chain + CompleteGraph + Ring)
   Age in (Old + Recent)
   Distance in (One + Two + Three + Four)
}

--                      CONFIGURATION

//  The maximalGraph of a network reflects the configuration.
fact {
  all network : Network {
    ( network.config = CompleteGraph
        =>
          all i, j : NodeId {
            i != j
              =>
                some link : network.maximalGraph {
                  (i + j) = link.ends
                }
          }
    )
    &&
```

```
    ( // This holds for up to (and including) 5 nodes.
      network.config = Ring
        =>
        all i  : NodeId {
          # { link : network.maximalGraph | i in link.ends } = 2
        }
    )
    &&
    ( // This holds for up to (but EXcluding) 5 nodes.
      network.config = Chain
        =>
          ( # { i : NodeId |
                # { link : network.maximalGraph | i in link.ends } = 1 }
                      = 2
            &&
            2 + # { i : NodeId |
                    # { link : network.maximalGraph
                                          | i in link.ends } = 2 }
                        = # NodeId
          )
    )
  }
}

--                      IMPLIED STATE COMPONENTS

// Some components of the state may be derived / infered from others
// but are recorded for convenience.  These include the
// believesConnectedTo, distance and onPathTo components of a node.

// The following definition for believesConnectedTo holds for
// up to (and including) 5 nodes.
fact {
  all node : Node {
    node.believesConnectedTo =
      { j : NodeId | j != node.id &&
         some link, link', link'', link''' : node.believesUp.link {
           (node.id + j) in
                (link.ends + link'.ends + link''.ends + link'''.ends)
           &&
           some ( link.ends & link'.ends )
           &&
           some ( link'.ends & link''.ends )
           &&
           some ( link''.ends & link'''.ends )
         }
      }
  }
}

// The set believesConnectedTo recorded by a node is the domain of the
// functions distance and onPathTo.
fact {
  all node : Node {
    (node.distance).Distance = node.believesConnectedTo
    (node.onPathTo).NodeId = node.believesConnectedTo
  }
```

```
}

// The distance function (maximum value Four for up to five nodes) is
// constructed as follows.
fact {
  all node : Node {
    all dest : node.believesConnectedTo {

      // A node is distance One only if there is a direct link to it.
      ( dest in (node.distance).One =>
        some link : node.believesUp.link { (dest + node.id) in link.ends }
      ) &&

      // A node is of distance Two only if it is connected
      // to a node of distance One.
      ( dest in (node.distance).Two =>
          some link : node.believesUp.link {
            dest in link.ends &&
            some (link.ends & (node.distance).One)
          }
      ) &&

      // A node is of distance Three only if it is connected
      // to a node of distance Two.
      ( dest in (node.distance).Three =>
          some link : node.believesUp.link {
            dest in link.ends &&
            some (link.ends & (node.distance).Two)
          }
      ) &&

      // A node is of distance Four only if it is connected
      // to a node of distance Three.
      ( dest in (node.distance).Four =>
          some link : node.believesUp.link {
            dest in link.ends &&
            some (link.ends & (node.distance).Three)
          }
      )
    }
  }
}


// The onPathTo function is constructed as follows
fact {
  all node : Node {

    // A node can only be the first node on the path to another node
    // if it is of distance One away.
    NodeId.(node.onPathTo) in (node.distance).One

    // A node of distance One away is the first node on the path to itself.
    all dest : (node.distance).One { dest.(node.onPathTo) = dest }

    // The first node on the path to a node of distance Two must be
    // directly connected to that node.
```

```
    all dest : (node.distance).Two {
      some link : node.believesUp.link {
        (link.ends - dest) in dest.(node.onPathTo)
      }
    }

    // The first node on the path to a node of distance Three or Four
    // must be indirectly connected to that node.
    all dest : (node.distance).Three {
      some link, link' : node.believesUp.link {
        ( some ( link.ends & link'.ends ) ) &&
        (link.ends - link'.ends) in dest.(node.onPathTo) &&
        (link'.ends - link.ends) = dest
      }
    }
    all dest : (node.distance).Four {
      some link, link', link'' : node.believesUp.link {
        ( some ( link.ends & link'.ends ) ) &&
        ( some ( link'.ends & link''.ends) ) &&
        (link.ends - link'.ends) in dest.(node.onPathTo) &&
        (link'.ends - link.ends) in link''.ends
        (link''.ends - link'.ends) = dest
      }
    }
  }
}

--                      FACTS

// Some facts about the beliefs of nodes.
fact {
  // Every node has a single belief about every link.
  all net : Network {
    all n : net.nodes {
      n.believesUp.link + n.believesDown.link = net.maximalGraph &&
      no (n.believesUp.link & n.believesDown.link)
    }
  }
  // No node has both old and new opinions about the status of a link.
  all n : Node {
    all timedLink, timedLink' : ( n.believesUp + n.believesDown ) {
      timedLink.link = timedLink'.link => timedLink = timedLink'
    }
  }
  // Knowledge of the status of adjacent links is always recent.
  all n : Node {
    all timedLink : (n.believesUp + n.believesDown) {
      n.id in timedLink.link.ends => timedLink.age in Recent
    }
  }
}

// Some facts about the network
fact {
  // No two nodes in a network have the same id
  all net : Network {
    all n, n' : net.nodes {
```

```
      n.id = n'.id => n = n'
    }
  }
  // Every node is included in the maximal graph of a network.
  all net : Network {
    net.nodes.id = NodeId
  }
}

--                      INITIAL STATE

// Initially all links are up and all nodes believe that all links
// are up.

fun InitialisedNetwork (network : Network) {
  network.upLinks = network.maximalGraph
  all n : network.nodes {
    InitialisedNode (n)
  }
  network.trace in EmptyTrace
}

fun InitialisedNode (node : Node) {
  no node.believesDown
  node.believesUp.age in Recent
}

--                      WORLD VIEW

// The WorldView records two networks: the initial network and the
// final one after some sequence of operations, as well as the trace that
// led from the first network to the second.
sig WorldView {
  initial, final : Network,
  makeLink, breakLink, propagate : Network -> Network
} {
  InitialisedNetwork(initial)
  all net, net' : Network {
    ( net' in makeLink[net] <=> MakeLink(net,net'))
    &&
    ( net' in breakLink[net] <=> BreakLink(net,net'))
    &&
    ( net' in propagate[net] <=> PropagateInfo(net,net'))

  }
  final in (^(makeLink + breakLink + propagate))[initial]
}

--                      FUNCTIONS OF THE SYSTEM

fun BreakLink (net,net' : Network) {
  // one link has been broken and the others remain unchanged.
  net'.upLinks in net.upLinks
  # net'.upLinks + 1 = # net.upLinks
  // the configuration and the maximal graph remain unchanged.
  net.config = net'.config && net.maximalGraph = net'.maximalGraph
  // the Break operation is added to the trace
```

```
  net'.trace.last in Break && net'.trace.prefix = net.trace
  // Observe that there is precisely one link in (net.upLinks - net'.upLinks)
  all breakingLink : (net.upLinks - net'.upLinks) {
    // the trace records which link was broken.
    net'.trace.last.link = breakingLink
    &&
    // The state of the nodes is changed as follows:
    all oldNode : net.nodes {
      some newNode : net'.nodes {
        // There is a new node for every old node.
        newNode.id = oldNode.id &&
        // Nodes not connected to the breaking link keep the same beliefs.
        ( newNode.id !in breakingLink.ends =>
            newNode.believesUp.link = oldNode.believesUp.link ) &&
        // Nodes connected to the breaking link change only their
        // belief about that link.
        ( newNode.id in breakingLink.ends =>
            newNode.believesUp.link =
              ( oldNode.believesUp.link - breakingLink ) ) &&
        // All beliefs about indirect links are now old.
        all timedLink : newNode.(believesUp + believesDown) {
          newNode.id !in timedLink.link.ends => timedLink.age in Old
        }
      }
    }
  }
}

fun MakeLink (net,net' : Network) {
  // one link has been broken and the others remain unchanged.
  net.upLinks in net'.upLinks
  # net.upLinks + 1 = # net'.upLinks
  // the configuration and the maximal graph remain unchanged.
  net.config = net'.config && net.maximalGraph = net'.maximalGraph
  // the Make operation is added to the trace
  net'.trace.last in Make && net'.trace.prefix = net.trace
  // Observe that there is precisely one link in (net'.upLinks - net.upLinks)
  all makingLink : (net'.upLinks - net.upLinks) {
    // the trace records which link was broken.
    net'.trace.last.link = makingLink
    &&
    // The state of the nodes is changed as follows:
    all oldNode : net.nodes {
      some newNode : net'.nodes {
        // There is a new node for every old node.
        newNode.id = oldNode.id &&
        // Nodes not connected to the making link keep the same beliefs.
        ( newNode.id !in makingLink.ends =>
            newNode.believesUp.link = oldNode.believesUp.link ) &&
        // Nodes connected to the breaking link change only their
        // belief about that link.
        ( newNode.id in makingLink.ends =>
            newNode.believesUp.link =
              ( oldNode.believesUp.link + makingLink ) ) &&
        // All beliefs about indirect links are now old.
        all timedLink : newNode.(believesUp + believesDown) {
          newNode.id !in timedLink.link.ends => timedLink.age in Old
```

```
        }
      }
    }
  }
}

// We now define a function that is true precisely when there is some
// information to be propagated.

fun NotOmniscient (net : Network) {
  some seeksInfo, hasInfo : net.nodes {
    // oldLink is a link of seeksInfo and recentLink is a link of hasInfo
    some oldLink : (seeksInfo.believesUp + seeksInfo.believesDown),
         recentLink : (hasInfo.believesUp + hasInfo.believesDown) {
      // they are talking about the same link...
      oldLink.link = recentLink.link &&
      /// but hasInfo has recent knowledge and seeksInfo does not
      oldLink.age in Old && recentLink.age in Recent &&
     // and hasInfo is first on seeksInfo's shortest path to one of
     // the line ends.
      hasInfo.id in (seeksInfo.onPathTo)[oldLink.link.ends]
    }
  }
}

fun PropagateInfo (net,net' : Network) {
  // There is some information to be propagated
  NotOmniscient(net)
  // No links are broken or made.
  net'.upLinks = net.upLinks
  // the configuration and the maximal graph remain unchanged.
  net.config = net'.config && net.maximalGraph = net'.maximalGraph
  // the Propagate operation is added to the trace
  net'.trace.last in Propagate && net'.trace.prefix = net.trace
  // Every node in the old system corresponds to a node in the new system.
  all oldNode : net.nodes {
    some newNode : net'.nodes {
      newNode.id = oldNode.id
    }
  }
  // At least one node learns some new information.
  some oldNode : net.nodes, newNode : net'.nodes {
    oldNode.id = newNode.id &&
    ( oldNode.believesUp != newNode.believesUp
      ||
      oldNode.believesDown != newNode.believesDown
    )
  }
  // The before state of a changed node is thisNode.
  // The after state of a changed node is thisNode'.
  all thisNode : (net.nodes - net'.nodes),
      thisNode' : (net'.nodes - net.nodes) {

    // We want to insist on the following constrainsts only if the
    // nodes have the same id, but we need to filter this additional
    // constraint through each quantification
```

```
    // Any newly learnt piece of information is recent
    all timedLink : ( (thisNode'.believesUp - thisNode.believesUp) +
                      (thisNode'.believesDown - thisNode.believesDown)) {
      thisNode'.id = thisNode.id =>
        timedLink.age in Recent
    }
    // Any overwritten piece of information must have been old.
    all timedLink : ( (thisNode.believesUp - thisNode'.believesUp) +
                      (thisNode.believesDown - thisNode'.believesDown)) {
      thisNode'.id = thisNode.id =>
        timedLink.age in Old
    }
    // A piece of information can be learnt only if a node on the
    // shortest path to the link had recent information about that link.
    all timedLink : (thisNode'.believesUp - thisNode.believesUp) {
      thisNode'.id = thisNode.id =>
        ( some neighbour : net.nodes {
            // the neighbour is on the shortest path to the link
            neighbour.id in (thisNode.onPathTo)[timedLink.link.ends] &&
            // this ensures that the information is recent.
            timedLink in neighbour.believesUp
          }
        )
    }
    all timedLink : (thisNode'.believesDown - thisNode.believesDown) {
      thisNode'.id = thisNode.id =>
        ( some neighbour : net.nodes {
            // the neighbour is on the shortest path to the link
            neighbour.id in (thisNode.onPathTo)[timedLink.link.ends] &&
            // this ensures that the information is recent.
            timedLink in neighbour.believesDown
          }
        )
    }
  }
}


--                       SEARCHING FOR INCONSISTENCIES

// We define a function Inconsistent that is true whenever two nodes
// in a network have conflicting opinions about the status of a link.

// The following works for up to a 4 node ring and a 4 node chain.
fun Inconsistent (net : Network) {
  some node, node' : net.nodes {
    // The nodes must be connected
    some link, link', link'' : net.upLinks {
      node.id in link.ends && node'.id in link''.ends &&
      some (link.ends & link'.ends) && some (link'.ends & link''.ends)
    }
    // They have different beliefs about the status of a link.
    some node.believesUp.link & node'.believesDown.link
  }
}

// We define a function Error that is true whenever a network is
```

```
// stable (ie there is no more information to be propagated) and there
// is an inconsistency.

fun Error (net : Network) {
  // There is no more information to be propagated
  ( ! NotOmniscient(net) )
  // But there is an inconsistency.
  Inconsistent(net)
}

// FUNCTIONS

// Some assertions about the state given that we are considering only
// 4 node rings.
fact {
  # NodeId = 4
  all n : Network {
    // Consider only rings.
    n.config in Ring
    // The maximal graph is A-B-C-D-A
    n.maximalGraph = LinkAB + LinkBC + LinkCD + LinkDA
--    // Dont remake any links.
--    no (n.trace.last & Make)
    // No node is distance Four from another node.
    no ( NodeId.(n.nodes.distance) & Four )
  }
}

fun FourNodeRingShowMeError (world : WorldView) {
  # NodeId = 4
  world.initial.config = Ring
  Error(world.final)
}

run FourNodeRingShowMeError for 3
  but 1 WorldView, 4 Distance, 2 Age, 4 Network, 4 Trace,
      2 Op, 4 NodeId, 14 Node, 4 Link, 8 TimedLink
  // 3 Configuration; 4 Distance; 2 Age - (since these are fixed)
  // 1 WorldView, 4 Network, 4 Trace, 2 Op
  // 4 NodeId, 14 Nodes, 4 Links, 8 TimedLinks
```

E Special-purpose Java program

```
public class RevPathFwd{

  // global variables
  static int N = 10; // number of nodes
  static float madeProb = (float) 0.3; // probability of each edge being made
  static int BROKEN = -1, UNKNOWN = 0, MADE = 1;

  static int actual[][] = new int[N][N];
  // actual[i][j] is actual cost of edge (i,j)

  static int believed[][][] = new int[N][N][N];
  // believed[k][i][j] is what k believes the cost of edge (i,j) to be;
  // believed[k] is symmetric; if k=i or k=j then believed[k][i][j] =
  // actual[i][j]

  static int provider[][] = new int[N][N];
  // provider[k][i] is the provider for i to k; calculated from believed[k]

  // We want to find values of actual, believed and provided such that if
  // provider[k][i] = p then believed[k][i][j] = believed[p][i][j] for all j>i

  // ---------------------------------------------------------------------
  public static void main(String argv[]){
    boolean done=false;
    int count=0;

    while(!done){
      done = tryGraph();
      count++;
      if(count%5000==0) System.out.println(count);
    }
  }

  // ---------------------------------------------------------------------
  // Generate graph
  static boolean tryGraph(){
    boolean ok;
    int i, j, k;

    // Initialise actual;
    do{
      for(i=0; i<N-1; i++)
        for(j=i+1; j<N; j++) actual[i][j] = actual[j][i] = random();

      // Check graph is connected
      ok = true;
      int[] prov0 = getProvider(0,actual);
      for(i=1; i<N; i++) if(prov0[i]==-1) ok = false;
      // if(!ok){System.out.println("Disconnected"); showGraph(actual); }
    } while(!ok);

    // Initialise believed
    for(k=0; k<N; k++)
      for(i=0; i<N; i++)
        for(j=i+1; j<N; j++)
```

```
        believed[k][i][j] = believed[k][j][i] =
          (k==i || k==j) ? actual[i][j] : UNKNOWN;

  // Try to choose beliefs so that condition is satisfied
  for(k=0; k<N && ok; k++){
    // Choose values for believed
    for(i=0; i<N; i++)
      for(j=i+1; j<N; j++)
        if(believed[k][i][j] == UNKNOWN)
          believed[k][i][j] = believed[k][j][i] = random();
    // System.out.println("Graph for "+k); showGraph(believed[k]);

    // Calculate providers
    provider[k] = getProvider(k, believed[k]);
    // System.out.println("Providers:"); showProvider(k); pause();

    // Test whether desired property is satisfied for this k
    for(i=0; i<N && ok; i++){
      int p = provider[k][i];
      if(p!=-1)
        for(j=0; j<N && ok; j++){
          // We want believed[k][i][j] == believed[p][i][j]
          if(believed[p][i][j]==UNKNOWN)
            believed[p][i][j] = believed[p][j][i] = believed[k][i][j];
          else
            ok = believed[k][i][j] == believed[p][i][j];
        }
    } // end for i
  } // end for k

  if(!ok) return(false);

  // Now check that beliefs differ somewhere
  boolean same = true;

  for(k=1; k<N && same; k++) // check if believed[k] = believed[0]
    for(i=0; i<N && same; i++)
      for(j=i+1; j<N && same; j++)
        same = believed[k][i][j] == believed[0][i][j];

  if(!same){
    showBeliefs(); System.out.println("Providers:"); showProviders();
  }
  else
    System.out.println("Same!");

  return(!same);
}

// ------------------------------------------------------------
// Generate random length
static java.util.Random rng = new java.util.Random();

static int random(){
  if(rng.nextFloat()<madeProb) return MADE; else return BROKEN;
}
```

```
// ------------------------------------------------------------
// get providers to k, based upon edge lengths in lengths; use Dijkstra's
// algorithm
static int[] getProvider(int k, int[][] lengths){
  int[] prev = new int[N], cost = new int[N], status = new int[N];
  int i;
  int INFINITY = 99999;
  int FIXED=1, PROVISIONAL=0; // status values
  int count=1; // number of FIXED nodes

  for(i=0; i<N; i++){
    prev[i] = k;
    cost[i] = i!=k ? (lengths[i][k]==MADE ? 1 : INFINITY) : 0;
    status[i] = i==k ? FIXED : PROVISIONAL;
  }

  while(count<N){
    // find minimum cost PROVISIONAL node
    int minNode=-1, minCost=INFINITY;
    for(i=0; i<N; i++)
      if(status[i]==PROVISIONAL && cost[i]<minCost){
        minCost = cost[i]; minNode = i;
      }

    if(minCost<INFINITY){
      // fix the status of this node
      status[minNode] = FIXED; count++;

      // update costs of all PROVISIONAL nodes
      for(i=0; i<N; i++)
        if(status[i]==PROVISIONAL && lengths[minNode][i]==MADE){
          int newCost = cost[minNode]+1;
          if (newCost<cost[i]){ cost[i] = newCost; prev[i] = minNode; }
        }
    }
    else count = N;
  } // end while(count<N)

  // calculate providers from prev
  int prov[] = new int[N];
  for(i=0; i<N; i++){
    if(cost[i]==INFINITY){
      // System.out.println("Disconnected: "+k+", "+i); showGraph(lengths);
      prov[i] = -1;
    }
    else{
      int j=i;
      while(prev[j]!=k) j = prev[j];
      prov[i] = j;
    }
  }

  return prov;
}

// ------------------------------------------------------------
static void showBeliefs(){
```

```
    for(int k=0; k<N; k++){
      System.out.println(k+": ");
      showGraph(believed[k]);
    }
  }

  static void showGraph(int[][] lengths){
    for(int i=0; i<N; i++)
      for(int j=i+1; j<N; j++)
        System.out.println(i+" "+j+" "+lengths[i][j]);
  }

  // Print all provider info
  static void showProviders(){
    for(int k=0; k<N; k++) showProvider(k);
  }

  // Print provider info for k
  static void showProvider(int k){
    for(int i=0; i<N; i++) System.out.println(k+" "+i+" "+provider[k][i]);
  }

}
```