XL: An Efficient Network Routing Algorithm

Kirill Levchenko
klevchen@cs.ucsd.edu

Geoffrey M. Voelker
voelker@cs.ucsd.edu

Ramamohan Paturi
paturi@cs.ucsd.edu

Stefan Savage
savage@cs.ucsd.edu

Department of Computer Science and Engineering
University of California, San Diego

ABSTRACT

In this paper, we present a new link-state routing algorithm called Approximate Link state (XL) aimed at increasing routing efficiency by suppressing updates from parts of the network. We prove that three simple criteria for update propagation are sufficient to guarantee soundness, completeness and bounded optimality for *any* such algorithm. We show, via simulation, that XL significantly outperforms standard link-state and distance vector algorithms—in some cases reducing overhead by more than an order of magnitude—while having negligible impact on path length. Finally, we argue that existing link-state protocols, such as OSPF, can incorporate XL routing in a backwards compatible and incrementally deployable fashion.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols

General Terms

Algorithms, Design, Theory, Performance, Experimentation

1. INTRODUCTION

"How do I best get from *here* to *there?*" This simple question is the essence of the routing problem, but it belies the considerable complexity embedded in modern intra-domain routing protocols. At the heart of this complexity is the issue of topology change. Routing in a static network is trivial, a simple table of directions calculated once for each destination. However, most real networks are dynamic—network links go up and down—and thus *some* nodes may need to be notified to recalculate their routes in response. This problem in turn can be boiled down to the question, "Who needs to know?" The traditional approach, enshrined in the family of *link-state* protocols, is to tell *everyone*; flood the topology change throughout the network and have each node then recompute its table of best routes. However as a network grows, this requirement to universally communicate and act on each topology change can become problematic. This is because a larger network also generates routing updates more often, necessitating more frequent route updates and route re-computation. Worse yet, these costs are incurred by every router in the network, meaning that the most resource-constrained router effectively determines the maximum network size that can be served by a routing algorithm. Thus, link-state protocols are frequently said to "not scale well."

However, it is manifestly unnecessary to communicate every link change to every router. Intuitively, only a small subset of router nodes are critically impacted by most link-state changes (particularly those whose shortest path trees include the changed link) and most other routing-related communication and computation is redundant. The traditional solution to this problem is to divide the network into separate routing domains and use this hierarchy to isolate topology updates. In the inter-domain context, the network is naturally divided into Autonomous Systems to reflect administrative and policy boundaries. However, the hierarchy imposed in the intra-domain context, for example with OSPF areas, is completely artificial: these areas do not delineate policy regions but rather serve as a routing algorithm optimization. As Cisco's OSPF Design Guide [6] states, "Areas are introduced to put a boundary on the explosion of link-state updates."

Unfortunately the process of properly configuring and maintaining areas is a complex art form; one with ad-hoc rules of thumb ("no more than 50 routers per area") and complex design tradeoffs.[1] Indeed, the structure imposed by areas inherently limits the kinds of topologies that can be mapped onto routes and, if not carefully managed, can produce arbitrarily sub-optimal routes and unnecessary points of failure [31]. Our work is focused on minimizing or removing the *need* for such artificial hierarchy by improving the efficiency of the underlying routing protocols.

Another approach to this problem is exemplified in the *fish-eye* routing optimization used by the 802.11s Mesh Networking standard. This technique simply limits the range over which topology updates are communicated, thus limiting updates to their immediate region [16, 14]. While this optimization imposes no operational burden, it is fundamentally unsound. Such protocols can neither guarantee that their routes will lead to their destinations (since they may contain loops) nor that all reachable destinations will have a valid route. While our work is motivated by the same desire to winnow update traffic, we seek to do so within the traditional constraints of correctness.

This state of affairs is fundamentally unsatisfying, and with link-state protocols being introduced into a wide range of new domains

SIGCOMM'08, August 17–22, 2008, Seattle, Washington, USA.

[1] In Moy's classic *OSPF: Anatomy of an Internet Routing Protocol*, he addresses the issue of how to place area boundaries as follows: "This is a complicated question, one without a single answer." and further clarifies that it depends on a combination of addressing structure, area size, topology considerations and policy considerations.

including overlay networks [2], ad-hoc and mesh networks [7], and to support traffic engineering for both MPLS [15] and Packet Based Backbone [9] technologies, we feel the issue is ripe for revisiting. To this end, our paper seeks to answer the following simple question: "Can one significantly increase routing protocol efficiency by selectively propagating topology updates, while still providing traditional guarantees of soundness, completeness and optimality?"

In addressing this question, this paper offers three contributions. First, we introduce the Approximate Link state (XL) routing algorithm, which can reduce routing overhead by an order of magnitude over existing protocols while still maintaining our correctness properties. Second, we show that three simple criteria for propagating updates are *sufficient* to ensure these properties for **any** link-state routing protocol:

S1 When the update is a cost increase (bad news),

S2 When the link is used in the node's shortest-path tree (propagated only to the next hop along the path to the link), and

C1 When it improves the cost to any destination by more than a $1 + \epsilon$ cost factor, where ϵ is a design parameter of the algorithm.

We show that all other updates may be safely suppressed. We show that these conditions are sufficient to guarantee that all forwarding paths are loop-free and within a $1 + \epsilon$ cost factor of optimal.

Finally, since our approach is primarily a *restriction* of the traditional link-state approach, it is possible to mix it within an existing link-state framework; allowing incremental deployment. We sketch how such interoperability could be achieved between native OSPF and a modified OSPF/XL protocol.

The remainder of the this paper is structured as follows: we briefly outline the relevant background and related work in Section 2, followed by a description of the network model and notation used throughout the paper in Section 3 and the XL routing algorithm itself in Section 4. Section 5 describes the simulation system we developed for evaluating the performance of routing algorithms. Then, in Section 6 we present our experimental evaluation the XL routing algorithm compared with link-state and distance-vector based approaches. In Section 7 we explain how OSPF may be modified to include the update suppression mechanism used in XL and Section 8 summarizes our results and concludes the paper.

2. BACKGROUND AND RELATED WORK

Beginning with the development of the ARPANET routing algorithms in the late seventies and early eighties [21, 22], network routing became a major area of research. The long-term loops suffered by the ARPANET distance-vector algorithm led to the development of link-state routing algorithms. In turn, a number of competitive distance vector algorithms were later developed that avoided long-term loops [4, 12, 17, 23, 28], including Garcia-Luna-Aceves' DUAL [10], which became the basis for Cisco's EIGRP [5]. To scale to larger networks, the link-state protocols OSPF and IS-IS introduced *area routing*. In this regime the network is manually divided into areas and while routing within an area takes place as before. Forwarding to destinations outside the local area is handled by special border routers—largely isolating most areas from the knowledge of any external topology change. As the OSPF specification states:

> [The] isolation of knowledge enables the protocol to effect a marked reduction in routing traffic as compared to treating the entire Autonomous System as a single OSPF domain. [24]

We are not the first to identify that areas can introduce problems in link-state networks. These problems have long been understood experimentally and are well summarized by AT&T's Mikkel Thorup in his "OSPF Areas Considered Harmful" [31]. Nor are we the first to look at reducing flooding overhead in link-state protocols. A number of such proposals have been made—typically for particular narrow regimes—including optimizations for flooding across interfaces [32], for reducing refresh overhead [27] and to damp the effects of route flapping [25]. We believe that our work is considerably more general than these efforts and with greater impact on efficiency.

Another approach to improving the scalability of link-state algorithms is the Link Vector (LV) algorithm introduced by Behrens and Garcia-Luna-Aceves [3]. The LV algorithm only propagates link updates about links in the node's shortest-path tree, an idea borrowed from distance vector algorithms, which we use in our work as well. However unlike our algorithm, the LV algorithm *explicitly* notifies neighbors when a link is added or removed from the shortest-path tree, whereas in our algorithm, the shortest-path tree is never explicitly communicated to neighbors; links not in the shortest-path tree are removed lazily only if their cost actually changes. This allows us to support approximation which, in turn, permits significant reductions in overhead for small increases in stretch, as our simulations show.

Finally, our notion of a *view* as a representation of network state is similar to that of Fayet *et al.* [8]. In their work, they give several sufficient conditions for routing in a network where nodes may have different views. However they do not give a routing algorithm or propose a mechanism for achieving these conditions.

3. DEFINITIONS AND NOTATION

In this section we formally describe our network representation and define what we mean by "forwarding." We then define the routing problem in terms of network configurations (e.g., "loop-free"). The reader may choose to skip directly to the ext section, where we describe the XL routing algorithm itself, turning back to this section for reference.

XL is a routing algorithm for a destination-based forwarding network such as the Internet. Formally, a *routing algorithm* is a mechanism by which network nodes can coordinate packet forwarding to ensure any two nodes in the network can communicate. In a *destination-based forwarding* network, forwarding is based on the packet destination address only. A node makes its forwarding decision using a *forwarding table* which either gives the next hop to each destination or indicates that the destination is not reachable by forwarding. The objective of a routing algorithm is to maintain a network configuration in which nodes are globally reachable by forwarding.

3.1 Network Model

We model the network as a graph $G = (V, E, e)$ with vertex set V, edge set E, and edge weight function e. The vertices represent network nodes, edges represent links, and edge weight represent link costs. Throughout the paper, we will use the pairs of terms node and vertex, link and edge, interchangeably.

To simplify exposition, the set of nodes and edges is fixed and globally known; only the edge weight function varies with time. It is straightforward to extend an algorithm in this model to allow vertices and edges to be inserted or deleted. The range of the weight function is the set of non-negative real numbers together with the special value ∞ having the usual semantics.

Let $n = |V|$, $m = |E|$ and let $N(u)$ denote the set of neighbors of $u \in V$. The set of edges E is undirected, however the weight function e *is* directed, which meaning that costs may be different along each direction of the link.

A path is a sequence of nodes of which any consecutive pair is adjacent in the graph. The *weight* of a path α in G, denoted $\|\alpha\|$ is sum of the weights (given by the weight function e) of its edges. Let $\delta(u, w)$ be the minimum weight of a path from u to w, or ∞ if no such path exists. If $\delta(u, w)$ is finite, we say that w is *reachable (in the network)* from u.

We use a superscript to denote the time at which the value of a function or variable is considered. For example, $\delta^t(u, w)$ denotes the weight of a minimum-weight path in G at time t. The domain of t is the set of non-negative real numbers. We say that a set of edges is *quiet* during a time interval if its weights do not change during the time interval. A set of edges *becomes quiet* at some time t if its edge weights do not change after time t.

3.2 Forwarding

To each node u in the graph we associate a *forwarding table* f_u which maps a destination node w to a neighbor of u, with the semantics that a packet arriving at u destined for w will be sent to the neighbor of u given by the forwarding table. If the packet has reached the destination or the destination is not reachable by forwarding, the forwarding table contains special value NONE. Thus,

$$f_u(w) \in N(u) \cup \{\text{NONE}\}, \tag{1}$$

where $N(u)$ are the neighbors of u.

We define the *configuration* of a forwarding network at some instant in time to be the set of all forwarding tables at that time. To capture the iterative nature of packet forwarding, we consider the path taken by a packet in the network. The *(instantaneous) forwarding path from u to w*, denoted $\phi(u, w)$, is the successive application of f to w, starting at u, up until NONE. Formally, $\phi(u, w)$ is the unique maximum-length sequence satisfying

$$\phi_0(u, w) = u \tag{2}$$

$$\phi_{i+1}(u, w) = f_{\phi_i(u,w)}(w) \tag{3}$$

$$\phi_{i+1}(u, w) \neq \text{NONE}. \tag{4}$$

Note that $\phi(u, w)$ may be an infinite sequence, (if for example $f_u(w) = v$ and $f_v(w) = u$) resulting in a *forwarding loop*. If $\phi(u, w)$ is a finite path from u to w, we say that w is *reachable by forwarding* from u.

3.3 Soundness and Completeness

To each node we associate a *routing process* responsible for computing the forwarding table of the node. The routing process knows (or measures directly) the costs of incident links and communicates with its neighbors via these links. A *routing algorithm* is the mechanism that defines what information is exchanged with neighbors and how the forwarding tables are computed. The central purpose of a routing algorithm is to maintain a forwarding configuration in which nodes are mutually reachable by forwarding. It is often also desirable for the paths taken by forwarded packets to be optimal or near-optimal. We formalize these objectives using the notions of *soundness*, *completeness* and *stretch*.

Definition. A configuration is *sound* if for all nodes u and w, $f_u(w) \neq$ NONE implies $\phi(u, w)$ is a path from u to w. A routing algorithm is *sound* if it produces a sound configuration after the network becomes quiet.

In a nutshell, soundness says that a node should only attempt to forward to destinations it can reach by forwarding. We will show that the XL routing algorithm we describe in this paper has this property. There is also a weaker property that is sufficient for many applications, and it is simply that there be no forwarding loops:

Definition. A configuration is *loop-free* if for all u and w, $\phi(u, w)$ is finite. A routing algorithm is *loop-free* if it produces a loop-free configuration after the network becomes quiet.

The difference between a sound and a loop-free configuration is that in the latter, a node only needs to know that forwarding to its next hop will not cause a loop (but the packet could be dropped somewhere down the path), while in a sound configuration, forwarding to the next hop must actually reach the destination.

The easiest way to achieve soundness is for every node to "pretend" everyone is unreachable by setting $f_u(w) =$ NONE for all destinations w. Clearly this is a degenerate configuration, so what we also want is for $f_u(w)$ to be NONE only if w really *is* unreachable from u in the network. We call this property *completeness*.

Definition. A configuration is *complete* if for all distinct u and w, $\delta(u, w) \neq \infty$ implies $f_u(w) \neq$ NONE. A routing algorithm is *complete* if it produces a complete configuration after the network becomes quiet.

Together the soundness and completeness properties say that all nodes are reachable by forwarding, but they say nothing about the optimality of the forwarding paths. This is the subject of our next definition.

Definition. The *stretch* of a configuration is the maximum taken over all distinct nodes u and w of the ratio $\|\phi(u, w)\|/\delta(u, w)$, with the convention that $1/\infty$ is 0, and ∞/∞ is undefined and not included in the maximum. A routing algorithm has *stretch* $1 + \epsilon$ if it produces a configuration with stretch at most $1 + \epsilon$ after the network becomes quiet.

4. THE XL ROUTING ALGORITHM

XL is fundamentally a link-state routing algorithm. It differs from the standard link-state algorithm in propagating only *some* link state updates. At the heart of the algorithm are three rules describing when an update should be propagated, and our main technical contribution is showing that these are *sufficient* for correctness as defined above. These conditions, which are at the heart of the algorithm, are:

S1 When the update is a cost increase (bad news),

S2 When the link is used in the node's shortest-path tree (propagated only to the next hop to the link), and

C1 When it improves the cost to any destination by more than a $1 + \epsilon$ cost factor, where ϵ is a design parameter of the algorithm.

Any updates not covered by the three rules above may be suppressed. The intuition behind these rules is that S1 and S2 ensure that each node's estimate of the distance to a destination decreases along the forwarding path, which ensures that no loops are formed. (More generally, S1 and S2 ensure *soundness* as described above.) Rule C1 ensures that all nodes know about some good (not but necessarily optimal) paths; this ensures *completeness* and bounded stretch. In the rest of this section, we formally describe our algorithm and describe how it implements these rules.

Because some updates are propagated while others are suppressed, nodes will not all have the same information about the network. To reason about this formally, we encapsulate a node's knowledge of the network in a *view*. A view is an edge weight function giving the weight of each edge at a particular point in time. Each node has an

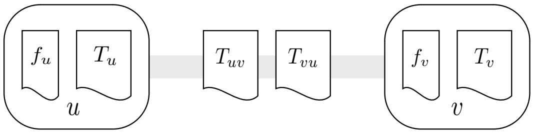


Figure 1: The routing process state for a pair of adjacent nodes. The routing process of each node maintains the forwarding table (f_u and f_v), internal view (T_u and T_v), and, for each neighbor, an external view (T_{uv} and T_{vu}). The forwarding table and internal view are private, while the external view T_{uv} can be atomically updated by u and atomically read by v. Similarly, the external view T_{vu} can be atomically updated by v and atomically read by u.

internal view containing the most recent edge weight information available to it. For each neighbor, a node also has an *external view*, which contains the edge cost information it wants to share with that neighbor. We denote the internal view of a node u by T_u and the external view of u for neighbor v by T_{uv}. For a pair of nodes u and v, their external views T_{uv} and T_{vu} will normally be the same, as the algorithm attempts to maintain "consensus" of external views. In describing the algorithm, we assume that the external view T_{uv} can be atomically written by u and atomically read by v. The forwarding table, internal view, and external views together constitute the state of the routing process (Figure 1).

Updating an external view incurs a communication cost, since the update must to be sent to corresponding neighbor. Our goal is to minimize the frequency of external view updates. To simplify analysis, we assume that external views can be updated even when the corresponding link has infinite cost. In practice, such updates would be queued until the link comes back up.

Formally, a view is a function mapping each edge to an *edge datum*, which is simply a pair of values p and t, written $p\,@\,t$, meaning that the edge had weight p at time t. Furthermore, views must only have correct information, meaning that the edge in question should have really had cost p at time t. We call this the *view invariant*. To avoid writing each definition twice, once for the internal views and once for external views, we will use the placeholder subscript $\diamond$ to mean both u and uv. With this convention, the view invariant is:

$$T_\diamond(x,y) = p\,@\,t \;\;\Rightarrow\;\; e^t(x,y) = p. \tag{V1}$$

For convenience, let $e_\diamond(x,y) = p$ denote the weight of (x,y) according to $T_\diamond$, that is, if $T_\diamond(x,y) = p\,@\,t$. But note that $e_\diamond$ is distinct from the true weight function e written with no subscript.

We say an edge datum $p\,@\,t$ is *more recent* than datum $p'\,@\,t'$ if $t > t'$. We will also use the terms *less recent* and *as recent* having the obvious meanings. Finally, we define a "most recent" operator "rec.“ Applied to a set of edge data S, $\operatorname{rec} S$ is the most recent datum in S. Formally, if there exists an edge datum $p\,@\,t \in S$ that is more recent than all other $p'\,@\,t' \in S$, then $\operatorname{rec} S = p\,@\,t$; otherwise, $\operatorname{rec} S$ is undefined.

Let $\pi_\diamond(z,w)$ be a minimum-cost path[2] from z to w in $T_\diamond$. Since the underlying graph is connected, such a path always exists, although the cost may not always have finite cost. Define $d_\diamond(w) = \|\pi_\diamond(u,w)\|_\diamond$; as before, $\diamond$ stands for both u and uv.

The routing algorithm is structured as an iterated state update algorithm. The process starts in the initial state defined by the initial views and then repeatedly executes the update algorithm, which updates the views and forwarding table. We start by defining the initial view.

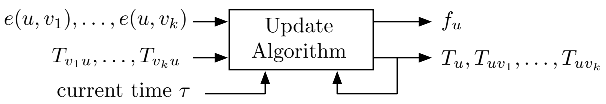

$e(u,v_1),\ldots,e(u,v_k)$
$T_{v_1u},\ldots,T_{v_ku}$
current time τ
Update Algorithm
f_u
$T_u, T_{uv_1},\ldots,T_{uv_k}$

Figure 2: The update algorithm computes the new forwarding table, internal view, and external views. The inputs to the algorithm are current incident edge weights, neighbors' external views, its previous internal view and external views. The algorithm also has access to the current time.

4.1 Initial View

The initial view defines the initial state of the routing process, before it has determined the incident link costs or communicated with its neighbors. In other words, it serves as the "base case" for the algorithm. The initial view, both internal and external, is defined as

$$T_\diamond(x,y) = \infty\,@\,0. \tag{5}$$

To satisfy the view invariant (Equation V1), we also define $e^0(x,y)$ to be ∞ for all $(x,y) \in E$.

4.2 Update Algorithm

The update algorithm computes a new forwarding table as well as new internal and external views. The input to the algorithm consists of the incident link costs, the current external views of its neighbors, and its own previous internal and external views, as well as the current time, denoted τ (Figure 2).

For the remainder of this section, fix a node u executing the update algorithm. The XL update algorithm has three phases. In the first phase, the algorithm computes a new internal view of u and the preliminary external views for its neighbors; in the second phase, it updates the forwarding table using the new internal view; and in the last phase, it computes new external views for each neighbor. We now describe these phases. Table 1 summarizes the notation used in the description and analysis of the routing algorithm.

4.2.1 Phase I: Internal and Prelim. External Views

τ	Time at the start of the iteration (INPUT).
$\epsilon_u(w)$	Maximum allowed relative error for destination w with respect to u (ALGORITHM PARAMETER).
T'_u, T'_{uv}	The internal view and external view for $v \in N(u)$, respectively, computed in the last iteration of the update algorithm, or, during the first iteration, the initial internal and external views (INPUT).
T_{vu}	The external view of $v \in N(u)$ (INPUT).
T_u, T_{uv}	The internal view and external view for $v \in N(u)$, respectively, currently being computed (OUTPUT).
T^*_{vu}	The preliminary external view of $v \in N(u)$ (Section 4.2.1).
f_u	The forwarding table of u, currently being computed (OUTPUT).
$e(x,y)$	Weight of edge (x,y) in G.
$e_\diamond(x,y)$	Weight of edge (x,y) in $T_\diamond$.
$\|\alpha\|$, $\|\alpha\|_\diamond$	Cost of path α in G and $T_\diamond$, respectively.
$\pi_\diamond(z,w)$	Shortest path from z to w in $T_\diamond$, with ties broken as consistently (Sections 4.2.2 and 4.2.3).
$d_\diamond(w)$	Cost of the shortest path from u to w in $T_\diamond$; by definition, $d_\diamond(w) = \|\pi_\diamond(u,w)\|_\diamond$ (Section 4.2.3).
$D_u(w)$	Minimum distance proxy from u to w (Section 4.4).

Table 1: Notation used in the description and analysis of the update algorithm. The symbol $\diamond$ represents the possible subscripts u or uv in the definitions.

[2] Ties may be broken arbitrarily, as long as the following consistency property is preserved: if $a\gamma b$ is a subsequence of $\pi_\diamond(z,w)$, then $\pi_\diamond(a,b) = a\gamma b$.

The first phase is concerned with view bookkeeping. Conceptually, we would like to have a single shared view for each pair of neighbors. However since the neighbors operate asynchronously, this would require a synchronization to ensure that the common view is updated correctly. Instead, we allow each neighbor to have its own version of this shared view. Neighbors keep their respective external views in agreement by only updating them with more recent information and by maintaining the invariant that a node's external view is no older than its neighbors. This ensures that the pair of views converge to the same single view. Thus first step in Phase I is to make sure the local external view is up to date with respect to the neighbor's external view for u. We call this updated view the *preliminary external* view. For each edge (x, y), the preliminary external view takes the more recent datum of the previous external view T'_{uv} and the neighbor's external view T_{vu}:

$$T^*_{uv}(x,y) = \text{rec}\left\{T'_{uv}(x,y),\, T_{vu}(x,y)\right\} \tag{6}$$

The preliminary external view is what the node and it's neighbor already agree on, or will agree on after the neighbor performs an update. It is the starting point for any updates the algorithm decided to communicate to its neighbor.

Next, we make the internal view the most recent information about each edge available to u. For edges incident on u, the most recent information is available locally and is only updated if the edge weight changes. Formally, for $v \in N(u)$,

$$T_u(u,v) = \begin{cases} e^\tau(u,v) \mathbin{@} \tau & \text{if } e^\tau(u,v) \neq e'_u(u,v), \\ T'_u(u,v) & \text{otherwise,} \end{cases} \tag{7}$$

where "rec" is the "most recent" operator.

For all other edges, the source of the most recent information are the external views. We collect the most recent datum for each edge. For all x and y where $x \neq u$,

$$T_u(x,y) = \operatorname*{rec}_v T^*_{uv}(x,y). \tag{8}$$

The following lemma follows by construction.

Lemma 1. *The internal view and preliminary external view are well-defined and satisfy the view invariant.*

4.2.2 Phase II: SPT and Forwarding Table

Having computed the internal view, which is the most recent information available to u about the state of the network, the update algorithm now computes a shortest-path tree using the internal view T_u and sets the forwarding table accordingly. This step is identical to the standard link-state algorithm.

Recall that $\pi_u(u, w)$ is a minimum-cost path from u to w in T_u, such that the set of all such paths from u forms a shortest-path tree. The distance from u to w in this tree is $d_u(w)$, which may be infinite if no finite-cost path exists. The forwarding table is now set according to the computed shortest-path tree: If $d_u(w) < \infty$ then set $f_u(w) = v$ where v is the next node in the path to w in the shortest-path tree; that is, where $\pi_u(u, w) = uv \cdots w$. Otherwise, if $d_u(w) = \infty$, set $f_u(w) = \text{NONE}$.

4.2.3 Phase III: External Views

In this last phase, the algorithm decides whether to propagate the latest datum to each of the neighbors. That is, for each neighbor v and each edge $(x, y) \in E$, the algorithm chooses whether to set $T_{uv}(x, y) = T_u(x, y)$, thereby propagating the new datum to v, or to set $T_{uv}(x, y)$ to $T^*_{uv}(x, y)$ suppressing the update. Recall that our goal is to bring the forwarding network into a sound and complete configuration with low stretch, as described in Section 3.3. We achieve these *global* objectives by enforcing the following three *local* constraints on external views.

The first two constraints, as we will soon show, guarantee soundness:

$$\forall (x,y) \in E \;\; e_{uv}(x,y) \geq e_u(x,y) \tag{S1}$$

$$\begin{aligned} \forall w \;\; \big(f_u(w) = v\big) \;\Rightarrow& \\ \forall (x,y) \in \pi_u(u,w) \;\; e_{uv}(x,y) &= e_u(x,y) \end{aligned} \tag{S2}$$

Constraint S1 states that we must never under-report an edge weight. This constraint ensures that in steady state all views reflect edge costs that are greater than or equal to the actual costs. Constraint S2 states that a node must advertise the latest edge cost to the neighbor v used to reach that edge. Intuitively, this constraint ensures that if v is our next hop to some destination w, then its own estimate of the distance to w will be no worse than ours, and, therefore, v will not attempt to reach w through us.

The third constraint guarantees completeness as well as bounded stretch. Before stating it, we need one more definition. Let $D_u(w)$ be a lower bound on the minimum distance from u to w in G. We show how $D_u(w)$ may be computed in Section 4.4. With these definitions in mind, the third constraint is:

$$\begin{aligned} \forall w \;\; d_{uv}(w) \leq \big(1 + \epsilon_u(w)\big) D_u(w) \;\; \text{or} \\ d_{uv}(u,w) = d_u(w). \end{aligned} \tag{C1}$$

It states that distances in the external view should not be much worse than actual. The lower bound $D_u(w)$ is used as a proxy for the actual distance $\delta(u, w)$.

It is possible to satisfy all three constraints by setting $T_{uv} = T_u$, that is, by propagating all edge datum updates. The resulting algorithm would behave exactly like the standard link-state algorithm. However by updating only the edges in the external view T_{uv} necessary to satisfy the constraints above, we can can reduce routing communication. The following algorithm does this.

Satisfying Constraints S1 and S2 is straightforward: an edge must be updated if it causes S1 or S2 to fail. Constraint C1 is more complicated.[3] Call an edge *hot*, denoted $\text{HOT}(x, y)$, if it lies on a path to a destination that causes Constraint C1 to fail.

$$\begin{aligned} \text{HOT}(x,y) = \exists w \;\; &\big((x,y) \in \pi_u(u,w)\big) \wedge \\ &\big(d_{uv}(w) > (1 + \epsilon_u(w)) D_u(w)\big). \end{aligned}$$

Our approach is to greedily update hot edges until Constraint C1 is satisfied. The complete update procedure in given in Algorithm 1.

It remains to show that Algorithm 1 produces an external view satisfying the Soundness and Completeness constraints above.

Lemma 2. *After executing Algorithm 1 (above) the external view T_{uv} satisfies the View Invariant V1 and Constraints S1, S2, and C1.*

4.3 Analysis

We now show that Constraints S1 and S2 produce a sound forwarding network configuration and Constraint C1 produces a complete configuration with bounded stretch. For the analysis, we assume that each execution of the update algorithm takes a bounded amount of time; let Δ be this duration. We will also need the following definition.

An edge (or set of edges) is *coherent* at a point in time if its associated external views are the same at that point in time. That is, an edge (u, v) is coherent at time t if $T^t_{uv} = T^t_{vu}$. Also, recall

[3] In fact, minimizing the number of edges that need to be updated to satisfy Constraint C1 is a hard problem (reduction from Set Cover).

Algorithm 1 PHASE III.

```
for all $(x, y) \in E$ do
    $T_{uv}(x, y) \leftarrow T^*_{uv}(x, y)$
    if $e_{uv}(x, y) < e_u(x, y)$ then
        $T_{uv}(x, y) \leftarrow T_u(x, y)$
    end if
    if $\big((x, y) \in \pi_u(u, y)\big) \wedge \big(f_u(y) = v\big)$ then
        $T_{uv}(x, y) \leftarrow T_u(x, y)$
    end if
end for
for all $(x, y) \in E$ do
    if HOT$(x, y)$ then
        $T_{uv}(x, y) \leftarrow T_u(x, y)$
    end if
end for
```

that a set of edges is *quiet* during a time interval if their weights do not change during the time interval.

Together the following two lemmas bound the cost of the forwarding path from u to w by $1 + \epsilon$ times the cost of the optimal path. Omitted proofs appear in the Appendix.

Lemma 3. *Fix a time $t > \Delta$. If $\phi^t(u, w)$ is a non-empty path that is both quiet during time interval $[t - \Delta, t]$ and coherent at time t, then $\phi^t(u, w)$ is a finite path from u to w and $\|\phi^t(u, w)\|^t \leq d^t_u(w)$.*

Lemma 4. *Fix a time $t > \Delta$. Let β be a path from u to w. If β is (i) quiet during $[t - \Delta, t]$, and (ii) coherent at time t, then*

$$d^t_u(w) \leq (1 + \epsilon)\|\beta\|^t,$$

where $\epsilon = \max_{x \in \beta} \epsilon_x(w)$.

Both Lemmas above are still conditioned on coherence. Here we show that a quiet network eventually becomes coherent, which will imply that our routing algorithm converges in finite time.

Lemma 5. *If a network is becomes quiet at some time t, then after a finite period of time it also becomes coherent.*

We can now state our main theorem.

Theorem 1. *If a network is quiet at and after some time t, then after a finite period of time the forwarding configuration becomes sound, complete, and has bounded distortion ϵ, where*

$$\epsilon = \max_{u,w} e_u(w).$$

Proof. By combining Lemmas 3, 4, and 5. □

4.4 Minimum Distance Proxy Function

Recall that the minimum distance proxy function D_u was used instead of the actual minimum distance function δ to define the Completeness constraint (C1) in Section 4.2.3 and was also used in Algorithm 1 to compute an external view. The correctness of the XL routing algorithm requires only that $0 \leq D_u(w) \leq \delta(u, w)$ for all u and w. However to give the algorithm leeway in suppressing updates, $D_u(w)$ should be as close to $\delta(u, w)$ as possible. Computing the exact distance $\delta(u, w)$ is exactly what we're trying to avoid by using approximation, so we choose $D_u(w)$ to be the distance computed by taking the weight of each edge to be the lowest cost of the edge ever observed. Because this value only changes when an edge cost drops below its all-time minimum cost, or an edge is added to the network, updates are infrequent and therefore introduce very little overhead to the algorithm. Furthermore, because all-time minimum link costs can only decrease, it can be computed using a distance vector-style algorithm without fear of loop formation, as shown by Jaffe and Moss [17].

A simpler alternative which does *not* guarantee globally bounded stretch is to set $D_u = d_u$. In other words, instead of computing and maintaining the cost lower bound as described above, we simply use out best estimate of the current cost from the internal view. In some cases, this will cause the stretch to exceed $1 + \epsilon$, although in practice the excess is likely to be quite small.

4.5 Cut Vertex Partitioning

Recall that in a *sound* configuration a node must only forward to a destination if the destination is reachable. This is hardly the case in the Internet today where ASes advertises prefixes, not individual destinations, even if part of the prefix is unreachable. For this reason, we introduced a weaker notion, that of a *loop-free* configuration, in which every forwarding path $\phi(u, w)$ must only be finite (loop-free) and not necessarily a path to the destination w. It means, essentially, that a node does not need to "know" that a destination is reachable before forwarding, only that forwarding to the next hop will not cause a loop. Practically, this means that sending a packet to an unreachable destination will generate an ICMP Unreachable message from a router further in the network rather than the local router.

As we have shown above, the basic XL algorithm is sound. If we relax the requirement of soundness, however, and settle for a loop-free algorithm, we can realize significant savings in routing communication using an extension to XL routing algorithm we call Cut Vertex Partitioning (CVP).

The idea behind CVP is based on the observation that a cut vertex, which is a vertex whose removal disconnects the graph, partitions the network graph into two or more separate subnetworks that can only communicate with each other through the cut vertex. This means that to communicate with a destination "across" a cut vertex, a node can simply forward to the cut vertex and it does not need to know about the network beyond the cut vertex. Thus with respect to routing, each subnetwork can be considered separately.

The CVP extension to the XL routing algorithm consists of the cut vertex forwarding policy described above, a mechanism for nodes to discover that they are cut vertices, and a cut vertex advertisement for nodes to learn which cut vertex to use to reach each destination. In our fixed, globally network model where only the edge weight function changes with time, all the necessary computation can be carried out by each node separately. In practice, however, where the topology is unknown and can change, cut vertex discovery and advertisement is slightly more involved; we do not describe it here.

In general, real networks do not have cut vertices that partition the network into large subnetworks where CVP could be used as a "divide and conquer" technique. However, what many real networks *do* have is a large number of leave. Since the neighbor of a leaf is necessarily a a cut vertex, CVP eliminates leaves from the routing computation, effectively reducing the size of the network. In fact, our implementation of CVP only considers such leaf cuts. Our experiments (Section 6) show that this "reduction by a thousand cuts" significantly decreases the communication load or routing.

5. THE SIMULATION SYSTEM

In this section we describe the simulation system we used to evaluate the performance of the XL routing algorithm. We designed our simulation system specifically for the purpose of evaluating the performance of routing algorithms on a forwarding network. At its heart is a discrete event simulator that simulates a number of routing algorithms including the XL algorithm. The simulation includes of the forwarding tables and all routing algorithm communication, but not other network traffic. It is expressly *not* a packet-level network simulator like `ns-2` and does not model or network characteristics such as packet loss, latency, or bandwidth.

The the core of the simulation system is the the `generator` program then generates an event script (a sequence of edge weight changes) for the simulation, and the `simulator` program that simulates a routing algorithm on the network using the generated event script. The output of the `simulator` program is a sequence of forwarding table updates. This sequence of processed by the analysis tools to compute convergence times, stretch, and related statistics.

5.1 Event Generator

The `generator` program produces a sequence of link cost changes according to a stochastic model of link failures. In the generated event sequence, a link is either *up*, in which case its cost is the nominal cost given defined by the weights file, or *down*, in which case its cost is ∞. The two directions are coordinated, that is, links (u, v) and (v, u) are either both up or both down.

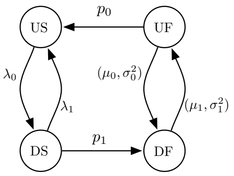


Figure 3: The link failure model used by the `generator` program. The up/stable, down/stable, up/flapping, and down/flapping states are denoted US, DS, UF, and DF, respectively.

Link failure and recovery is controlled by a stochastic process (Fig. 3). Each link is treated independently. In addition to being up or down, a link is also either *stable* or *flapping*. The four link-states are thus up/stable, down/stable, up/flapping, and down/flapping. In the stable state, the link time-to-failure is distributed exponentially with mean λ_0. Once down, a link may remain in the down/stable state, in which case the time-to-recovery is distributed exponentially with mean λ_1, or, with probability p_1 a link may become unstable and transition to the flapping/down state. Thus, parameter p_1 controls the propensity of links to flap. In the flapping state, the time-to-recovery has a normal distribution truncated to $[0, \infty)$ with parameters μ_1 and σ_1^2, and time-to-failure has a similarly truncated normal distribution with parameters μ_0 and σ_0^2. After recovering from failure in the flapping state a link leaves the flapping state with probability p_0. Parameter p_0 thus controls how long a link remains flapping.

Our link event model is a generalization the two-state model of Park and Corson [26]; we added the flapping failure mode, which we expected the XL algorithm handle particularly well. When $p_1 = 0$, link failures are independent with exponentially-distributed failure and recovery times. On the other hand, when $p_1 = 1$, all links have an exponentially distributed time-to-first-failure followed by repeated up-down cycles controlled by the p_0 parameter.

5.2 Protocol Simulator

The `simulator` program is a discrete event simulator that simulates a single routing algorithm under a given topology and link event sequence. In other words, it simulates n instances of the routing algorithm running in parallel, one on each node. The simulator repeatedly executes the update algorithm of each node, providing as input the (simulation) time at the start and end of the current iteration of the algorithm, the costs of incident links, and its message queue, consisting of messages sent by its neighbors since the last invocation of the update algorithm on this node. The update algorithm performs any processing dictated by the algorithm, and if necessary, updates its forwarding table and then posts messages to its neighbors. The (simulated) duration of the iteration is chosen randomly according to a normal distribution truncated to $[0, \infty)$ with parameters $\mu_\triangle$ and $\sigma_\triangle^2$; we chose the normal distribution because it was familiar and because the model did not seem unreasonable to us.

Name	n	m	D_1	D_2	D_3	Description
CROWN X	$3X$	$4X$	0	1/3	2/3	Two cycles of size X and $2X$ with nodes in the smaller connected to alternate nodes in the larger.
HONEY	—	—	0	~ 0	~ 1	A hexagonal grid.
QUAD	—	—	0	~ 0	~ 0	A rectangular grid.
ABILENE	11	14	0	45%	55%	Abilene with routing metrics [1].
ARPANET	59	72	7%	48%	41%	ARPANET (March 1977) [11].
FUEL1221	104	151	49%	19%	6%	AS 1221 from RocketFuel [19].
FUEL1239	315	972	10%	19%	16%	AS 1239 from RocketFuel [19].
F. 1221C	50	97	0	50%	6%	The 2-core of FUEL1221.
F. 1239C	284	941	0	22%	18%	The 2-core of FUEL1239.
ORB145	145	227	29%	28%	17%	FUEL1239 rescaled (`-n 200`).
ORB257	257	433	31%	20%	21%	FUEL1239 rescaled (`-n 300`).
ORB342	342	606	33%	24%	14%	FUEL1239 rescaled (`-n 400`).
ORB406	406	791	27%	28%	14%	FUEL1239 rescaled (`-n 500`).
ORB497	497	961	29%	26%	17%	FUEL1239 rescaled (`-n 600`).
ORB575	575	1081	31%	25%	16%	FUEL1239 rescaled (`-n 700`).
ORB664	664	1300	26%	27%	17%	FUEL1239 rescaled (`-n 800`).
ORB729	729	1427	32%	24%	16%	FUEL1239 rescaled (`-n 900`).
ORB813	813	1584	29%	25%	16%	FUEL1239 rescaled (`-n 1000`).
ORB892	892	1694	34%	26%	15%	FUEL1239 rescaled (`-n 1100`).

Table 2: Network topologies used in the experiments. Column legend: n – number of nodes; m – number of links; D_1, D_2, and D_3 fraction of nodes of degree 1, 2, and 3, respectively. All but the FUEL networks have unit link costs.

The `simulator` program contains implementations of the following routing algorithms.

`ls` The standard link-state algorithm [22] which is the basis for OSPF and IS-IS.

`dv` A distance vector algorithm very similar to RIP [20] with split horizon. The maximum distance bound is a global parameter of the algorithm.

`dv+p` A modern distance vector algorithm which uses a parent pointer to detect loops [4, 12, 28].

`lv` The Link Vector algorithm proposed by Behrens and Garcia-Luna-Aceves [3].

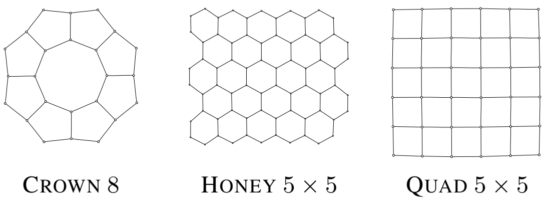
`xl` The XL algorithm described in this paper, parametrized by error ϵ. When $\epsilon = 0$, all forwarding paths are optimal just as with the above algorithms.

All of the above algorithms send updates only when a topology change occurs (sometimes called "triggered update"), and there are no periodic updates.

The output of the simulation is a sequence of forwarding table updates written to the `update` file for later processing. At the end of the simulation, the `simulator` program reports the total number of messages and bytes sent by the routing processes as well as the maximum messages and bytes sent by a single node.

6. EVALUATION

In this section we experimentally evaluate the performance of



CROWN 8   HONEY 5×5 QUAD 5×5

Figure 4: Small examples of synthetic networks.

the XL routing algorithm relative to existing routing algorithms. Our objective is to evaluate the claims that the XL routing algorithm:

- Sends fewer routing updates,
- Does not significantly sacrifice correctness, convergence time, or stretch, and
- Continues to perform well as the network grows.

Our evaluation is based on simulations of the four protocols implemented by the `simulator` program (`ls`, `dv`, `dv+p`, and `xl`) on a number of networks and under two different link event models. The main result of simulation is that the XL routing protocol does indeed reduce the number of updates: compared to the link-state algorithm, XL generates between 2 and 20 times fewer updates (Table 4). This experiment is discussed in Section 6.2; first, however, we describe our experimental setup.

6.1 Experimental Setup

Each experiment consists of a number of simulation runs. Each run simulates a single routing algorithm for 86,400 seconds (one day) at a rate of 10 iterations of the update algorithm per second.

Networks. We used the following networks in our simulations: three synthetic networks, the Abilene backbone [1], the ARPANET topology from March 1977 [11], two Rocketfuel networks with inferred link costs [19], and a series of networks created by rescaling the Sprint network (AS 1239) from the Rocketfuel dataset using Orbis [18]. The Orbis command-line arguments to the `dkRescale` program were "`-k 1 -n` n_{nom}", where the nominal size n_{nom} ranged from 200 to 1100. Table 2 describes the networks used in the experiments and Figure 4 shows small instances of synthetic networks. The synthetic networks allowed us to test the routing algorithms on topologies based design decisions different from the AS router-level topologies. In particular, the large-diameter HONEY and GRID networks shed some light on how the algorithms might perform in wireless ad-hoc networks.

We also created the 2-cores of the two Rocketfuel networks. The 2-core of a graph is the graph resulting from repeatedly removing all degree-1 nodes [29]. With no degree-1 nodes, CVP (which was implemented only for leaf nodes) would have no effect, allowing us to also evaluate the value of this optimization.

Link Events. All link events for the simulation were generated using the `generator` program (Section 5.1). Recall that in the `generator` link event model, a link is either up (nominal weight) or down (infinite weight); the time between failures and failure duration are controlled by the four-state stochastic model shown in Figure 3. In our simulation, we used two different sets of model parameters: a Standard set in which a link fails about once a day, and comes back up in about an hour, and the Flapping set in which links are less likely to fail, but more likely to fail repeatedly (flap); Table 3 gives the precise model parameters.

Both the Standard model and Flapping model are more aggressive that what might be expected of a real network [13, 30]. We wanted to stress the routing algorithms under the kinds of conditions where routing algorithm efficiency matters greatly, namely where many links are unstable (Standard model) or only some are unstable but tend to oscillate (Flapping model).

Algorithm Parameters. The distance vector algorithm (`dv`) requires a maximum distance bound (the so-called "infinity metric") to detect routing loops. For the simulations, this value was computed by using a linear program to approximate the cost of the longest path. The XL routing algorithm (`xl`) has an error parameter ϵ that determines the stretch. In the experiments, we simulated `xl` with $\epsilon = 0.0$ and $\epsilon = 0.5$, corresponding to no stretch and a maximum stretch of 1.5. Increasing ϵ beyond 0.5 did not appear to significantly reduce the number of updates generated by the algorithm beyond the $\epsilon = 0.5$ level.

6.2 Performance

In this section we evaluate our first two claims: that compared to existing routing algorithms, the XL algorithm uses fewer updates to achieve comparable performance. We simulated each routing algorithm on the synthetic and measured topologies. Each combination of algorithm, network and link event model (Standard or Flapping) was simulated 10 times and averaged in reporting results. For each combination, the 10 simulations differed only in the link events.

Total Communication. Table 4 shows the average number of messages sent during the simulation relative to `ls`, the link state algorithm, which provides a convenient baseline for comparison.

Referring to the table, the most erratic performer was `dv`, which was highly sensitive to topology: it did extremely well on networks such as QUAD 16×16 with many equal-cost paths and poorly on networks with long cycles that trigger its "counting-to-infinity" behavior. As expected, both `dv+p` and `lv` performed similarly: they routinely did better than `ls` but could not take advantage of the multiple equal-cost paths in QUAD networks as well as `dv` did.

The XL algorithm performed consistently well on all networks. Like `dv`, it was able to take advantage of path redundancy in the QUAD synthetic network. It also did well on "leafy" networks like FUEL1221, where CVP played a major role in reducing communication.

We note that XL algorithm performed particularly well in the flapping model. Why is this? The reason is that the XL algorithm

	p_0	p_1	λ_0^{-1}	λ_1^{-1}	μ_0	σ_0	μ_1	σ_1
Standard	0.25	0.10	1 d	1 h	1 m	10 s	1 m	10 s
Flapping	0.25	1.00	2 d	10 s	10 s	1 s	10 s	1 s

Table 3: Parameters used to generate link events according to the `generator` link event model described in Section 5.1. Mean time-to-failure is controlled by the λ_0^{-1} parameter and the probability of a repeat failure by the p_1 parameter. Units: d – days, h – hours, m – minutes, s – seconds.

	Standard model					Flapping model				
	`dv`	`dv+p`	`lv`	`xl`		`dv`	`dv+p`	`lv`	`xl`	
CROWN 64	3.13	1.11	1.10	0.64	0.41	0.85	0.82	0.82	0.45	0.11
H. 16×16	0.95	0.69	0.65	0.31	0.18	0.28	0.65	0.60	0.20	0.06
Q. 16×16	0.12	0.40	0.39	0.14	0.10	0.06	0.38	0.37	0.07	0.04
ABILENE	0.82	0.71	0.71	0.50	0.43	0.88	0.79	0.79	0.47	0.33
ARPANET	2.33	1.02	1.02	0.47	0.40	1.80	1.00	0.99	0.36	0.24
FUEL1221	7.90	0.63	0.62	0.14	0.10	7.05	0.61	0.60	0.12	0.05
FUEL1239	5.01	0.25	0.26	0.17	0.09	1.21	0.25	0.25	0.14	0.04
F. 1221C	0.79	0.45	0.46	0.34	0.22	0.39	0.42	0.42	0.27	0.11
F. 1239C	0.99	0.25	0.25	0.19	0.09	0.21	0.24	0.24	0.14	0.04

Table 4: Average number of messages after initialization, relative to `ls` (average of 10 simulation runs). The `xl` columns shows values for algorithm parameters $\epsilon = 0.0$ (first value) and $\epsilon = 0.5$ (second value).

	Standard model					Flapping model				
	`dv`	`dv+p`	`lv`	`xl`		`dv`	`dv+p`	`lv`	`xl`	
Crown 64	3.41	1.07	1.06	0.68	0.46	1.09	0.79	0.78	0.49	0.17
H. 16 × 16	1.09	0.73	0.68	0.35	0.23	0.42	0.71	0.64	0.24	0.09
Q. 16 × 16	0.16	0.45	0.43	0.18	0.14	0.12	0.44	0.42	0.10	0.07
Abilene	0.97	0.77	0.77	0.64	0.55	0.98	0.83	0.83	0.55	0.46
Arpanet	2.28	0.91	0.89	0.51	0.45	1.86	0.89	0.87	0.39	0.28
Fuel1221	7.32	0.46	0.46	0.12	0.09	6.56	0.44	0.43	0.10	0.05
Fuel1239	4.85	0.23	0.23	0.20	0.11	1.16	0.21	0.21	0.16	0.05
F. 1221c	0.74	0.38	0.38	0.37	0.26	0.34	0.35	0.36	0.30	0.16
F. 1239c	0.95	0.22	0.22	0.22	0.11	0.20	0.22	0.21	0.17	0.05

Table 5: Average (over 10 simulations) of the maximum number of messages generated by any one node, relative to `ls`. The `xl` columns shows values for algorithm parameters $\epsilon = 0.0$ (first value) and $\epsilon = 0.5$ (second value).

tends to move away from flapping links: The the first time a link fails, an update is sent to all nodes in whose shortest-path tree it appears, that is, nodes that used the link to reach some destination. When the same link comes back up, many of the nodes which used it keep their current path because it is only slightly worse than the previous path which used the link. As a result, fewer nodes now have the link in the shortest-path tree, so that when it fails again, they are not affected. Thus, after the first failure, the effects of the link are generally limited to a small neighborhood around the link where the link is a significant fraction of path costs.

Per-Node Communication. Table 5 shows the maximum number of messages generated by any single node during the simulation, relative to `ls`. In contrast to the total communication, this number shows the maximum load placed on an *individual* node rather than the network as a whole. Although it is does not show short-term load on a node, it does show whether a routing algorithm spreads the communication costs evenly across the network or whether it creates bottleneck routers.

These results do not differ markedly from the total communication results shown in Table 4, indicating that none of the algorithms loaded any one node significantly more heavily than the link-state algorithm, in which the number of messages sent by a node is proportional to its degree.

Stretch. In addition to counting the number of messages, we performed additional analysis as described in in Section 5. The first quantity we consider is stretch; recall that stretch is the ratio of the forwarding cost to optimal cost between a pair of nodes. Because stretch is an instantaneous measure for each pair, it is not an easy value to summarize for an entire simulation. We use the top stretch centile for each pair. By the top centile, we mean the lowest upper bound for 99% of the simulation duration. In other words, a pair's stretch is at most the top centile value 99% of the time. In Table 6 we report the median, average and maximum top centile stretch over all pairs for `xl` with parameter $\epsilon = 0.5$, corresponding to maximum allowed stretch of 1.5. For all other algorithms, including `xl` with $\epsilon = 0.0$, the maximum top centile stretch was zero as expected, and is not shown.

Clearly, while the stretch approaches the maximum 1.5 for some source-destination pairs, the average stretch is quite good, in all cases at most 5% optimal. In fact, since the median is 1.00, for the majority of nodes the forwarding path is optimal. By just *allowing* the XL algorithm to choose sub-optimal paths we were able to get the reduction in communication complexity while paying only a fraction of the allowed 50% penalty.

Convergence. Finally, we consider the convergence time of the XL routing algorithm. By "convergence time" we mean the time it takes a routing algorithm to establish a desirable (e.g., sound, complete) forwarding configuration. In essence, it combines the time it takes a routing algorithm to re-establish a sound (or loop-free) configuration after a link failure and the time it takes the algorithm to start using a lower-cost path when it becomes available.

	Standard model			Flapping model		
	Med	Avg	Max	Med	Avg	Max
Crown 64	1.00	1.02	1.43	1.00	1.01	1.39
H. 16 × 16	1.00	1.05	1.45	1.00	1.02	1.44
Q. 16 × 16	1.00	1.02	1.43	1.00	1.01	1.40
Abilene	1.00	1.01	1.22	1.00	1.01	1.18
Arpanet	1.00	1.02	1.45	1.00	1.01	1.41
Fuel1221	1.00	1.01	1.34	1.00	1.01	1.33
Fuel1239	1.00	1.04	1.41	1.00	1.02	1.41
Fuel1221c	1.00	1.02	1.35	1.00	1.01	1.33
Fuel1239c	1.00	1.04	1.42	1.00	1.02	1.41

Table 6: Top centile stretch for `xl` with parameter $\epsilon = 0.5$. The median, average, and maximum of the top centile were taken over all source-destination pairs; a pair's instantaneous stretch is at most its top centile value 99% of the time.

	Standard model					Flapping model				
	`dv`	`dv+p`	`lv`	`xl`		`dv`	`dv+p`	`lv`	`xl`	
Crown 64	4.08	0.00	0.00	1.04	0.88	9.28	0.00	0.00	1.17	0.66
H. 16 × 16	17.19	0.00	0.00	0.99	0.88	1.49	0.00	0.00	0.90	0.80
Q. 16 × 16	5.96	0.00	0.00	1.00	0.98	1.24	0.00	0.00	1.16	1.03
Abilene	2.27	0.00	0.00	0.79	0.87	1.83	0.00	0.00	0.93	0.98
Arpanet	3.12	0.00	0.00	0.91	0.82	2.86	0.00	0.00	0.94	0.82
Fuel1221	74.23	0.00	0.00	0.79	0.79	46.01	0.00	0.00	0.79	0.81
Fuel1239	85.64	0.00	0.00	0.92	0.87	24.87	0.00	0.00	0.95	0.85
F. 1221c	10.80	0.00	0.00	0.87	0.85	2.60	0.00	0.00	0.96	0.95
F. 1239c	25.12	0.00	0.00	0.95	0.86	2.24	0.00	0.00	0.99	0.85

Table 7: Forwarding loop duration maximum over all source-destination pairs, relative to `ls`. The forwarding loop duration for a pair of nodes u and w is the duration of time $\phi(u, w)$ was infinite.

	Standard model					Flapping model				
	`dv`	`dv+p`	`lv`	`xl`		`dv`	`dv+p`	`lv`	`xl`	
Crown 64	2.58	2.74	2.73	1.54	1.74	5.29	5.44	5.37	1.45	1.41
H. 16 × 16	1.19	3.08	2.46	1.10	1.09	1.30	4.85	3.12	1.02	0.93
Q. 16 × 16	1.10	2.54	2.00	1.03	1.03	1.02	2.92	2.12	0.99	0.99
Abilene	1.25	1.41	1.41	1.05	1.14	1.36	1.55	1.56	1.01	1.02
Arpanet	1.29	1.41	1.34	0.95	0.94	1.20	1.48	1.46	0.96	0.89
Fuel1221	1.04	1.15	1.09	0.60	0.63	1.06	1.16	1.14	0.52	0.52
Fuel1239	1.15	1.44	1.36	0.75	0.76	1.04	1.24	1.22	0.74	0.70
F. 1221c	1.16	1.38	1.36	1.03	1.09	1.33	1.62	1.41	1.00	0.98
F. 1239c	1.54	1.76	1.57	1.05	1.03	1.50	1.70	1.63	1.01	0.93

Table 8: Maximum duration of infinite forwarding-to-optimal distance ratio relative to `ls`. The maximum is taken over all source-destination pairs. The infinite forwarding to optimal distance ratio duration for a pair of nodes u and w is the duration of time when $\|\phi(u, w)\|$ was infinite but $\delta(u, w)$ was not.

The `analyzer` program does not measure convergence time directly; instead, it measures the duration of forwarding loops and the time to establish a new forwarding path when a node becomes reachable. The former is reported in Table 7 as the maximum, over all source-destination pairs, of the combined duration of forwarding loops. The time to establish a new forwarding path is reported in Table 8 as the maximum, over all source-destination pairs, of the total time the forwarding distance was infinite while the optimal distance was not. In both tables, results are shown relative to `ls`.

It comes as no surprise that the generic distance vector algorithm has a problem with long-lasting loops. In contrast, loops in `dv+p` and `lv` are extremely rare and short-lived because, although it is not guaranteed loop-free at all times, its policy for accepting a next hop are fairly conservative. The same "reluctance" to accept a new path

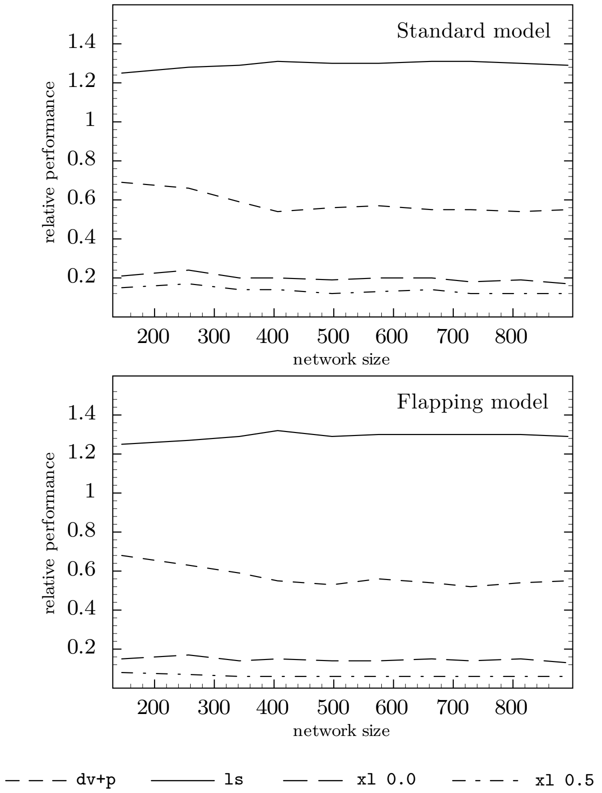


Figure 5: Number of messages as a function of network size for the ORB family of networks; values normalized by number of edges in the graph. Both dv+p and lv performed similarly (within 5%); only dv+p is shown. The distance vector algorithm was omitted because its communication exceeded the other algorithms by a factor of 5 in the Standard model and nearly an order of magnitude on the Flapping model.

is also responsible for the longer time to establish a new forwarding path, although `lv` seemed to have slightly faster convergence.

With the exception of the CROWN network, `xl` had slightly better convergence times than `ls`. This is because `xl` changes its next hop to a destination only if it is much better than the current next hop, thus updating the forwarding table less often and avoiding short-term loops or unreachable configurations. On the other hand, the time to accept a new forwarding path is generally longer than `ls` because `xl` has less information about the network, so that when a link fails, it may be necessary for the link failure update to propagate before a bypass route is advertised. CVP partially remedies this the situation because when a cut edge comes up, only the corresponding cut vertices need to be updated to restore the path.

6.3 Scalability

To evaluate the scalability of the XL routing algorithm relative to existing algorithms, we simulated each algorithm on families of networks of increasing size: the HONEY synthetic network family and the ORB re-scaled network family described earlier. Each combination of algorithm, network, and link event model (Standard and Flapping) was simulated 5 times and averaged in reporting results. Figure 5 shows algorithm communication as a function of network size for the ORB family of networks. Except for `dv`, results on synthetic networks was similar; `dv` performance was highly variable from one family to another.

As the network size increases, `xl` maintains its good relative performance. As with other algorithms, however, the routing communication load still grows linearly with the size of the network. This is because a link failure still triggers partial flooding to nodes whose shortest-path tree included the failed link, and roughly half of all simulation events are link failures. In a connected network, a node's shortest-path tree contains $n - 1$ nodes, so the probability of a node being affected by a link failure is $(n - 1)/m$, and thus the expected number of nodes affected by a random link failure is about n^2/m. This means that in a network such as the Internet where m/n is small, a random link failure will be propagated to a constant fraction of the nodes.

7. OSPF WITH XL

This section is motivated by the observation that the XL routing algorithm and the standard link-state algorithm are inherently compatible. This is because flooding satisfies Conditions S1, S2, and C1, so it is possible to mix instances of XL and the standard link-state algorithm. In this section, we sketch how the routing algorithm used with the OSPF Version 2 protocol [24] can be modified to take advantage XL's update suppression mechanism, while still remaining compatible with the original OSPF. In other words, routers running the modified algorithm, which we call OSPF/XL, can inter-operate in a mixed-deployment scenario with those running the standard OSPF algorithm. We emphasize, however, that we have not implemented these modifications and that all our evaluations are based on simulation at this point. We leave implementing OSPF/XL to future work, although we do not believe it should be too challenging.

Recall that in the XL algorithm the state of the network consists of the internal and external views. The internal view already exists in OSPF as the link-state table. External views, however, have no OSPF analog. To save memory, we suggest that external views should not be materialized, rather, they can be represented as differences from the internal view. Since a node's internal and external views will typically contain a lot of the same information, we de no expect the additional memory required for external views to be significant.

The second modification to OSPF is in the way updates are processed. Upon receiving an update, a node records it in the external view of its incoming interface. If the update has newer information than in the internal view, the internal view is updated as well. Next, the main shortest-path tree is re-computed from the internal view. Algorithm 1 is then used to update other external views and determine to which interfaces the update should be propagated. Periodically, not necessarily after each update, the main shortest-path tree is used to update the forwarding table.

Finally, the proxy minimum distance $D_u(w)$ used in Algorithm 1 will need to be approximated. The easiest way to do this is for each node to simply keep a record of the smallest distance to each destination observed during some period of time, say 1 day, and use this value instead. We believe that such an approximation is adequate in all but the worst pathological cases.

Overall, OSPF/XL requires only modest changes to the standard OSPF in order to take advantage of our update suppression mechanism. Moreover, the benefits of XL can be realized even in a mixed environment where only some of the routers implement OSPF/XL—incentivizing incremental deployment.

8. CONCLUSION

We have presented the XL routing algorithm, a new link-state routing algorithm specifically designed to minimize network communication. XL works by propagating only *some* of the link-state updates it receives, thereby reducing the frequency of routing updates in the network. We also formally proved the correctness of

XL and validated our performance claims in simulation. In particular, our simulation showed that with a small penalty in stretch, our algorithm dramatically reduced the number of updates needing to be communicated and processed.

However, in allowing the routing algorithm to choose slightly sub-optimal routes, the network operator also cedes some degree of control. In particular, traffic engineering via link costs is harder since *current* traffic forwarding will be determined, in part, by *past* link costs. Fortunately, it is easy to augment our algorithm to "flush" all suppressed updates periodically, causing it to propagate and use exact routing information. In fact, the approximation parameter ϵ can be adjusted dynamically in response to load. By setting $\epsilon = 0$ locally under normal conditions and and $\epsilon = 0.5$ under load or in the presence of flapping, the network can achieve the best of both worlds: deterministic routing in normal circumstances, approximate routing under heavy load.

Finally, we also believe that there may be significant opportunities to improve the efficiency of link state routing even further. In particular, recall that the XL routing algorithm propagates all link cost *increase* updates, meaning that, on average, it will propagate half of all updates that affect it. It is natural to ask whether this is strictly necessary, or whether a superior algorithm—one that selectively suppresses link failures—can scale *sub-linearly* for typical networks. Whether such an algorithm exists and can guarantee soundness and correctness remains an open problem that we hope to address in future work.

9. ACKNOWLEDGEMENTS

This research was supported in part by National Science Foundation grants NSF-0433668 (CCIED) and EIA-0303622 (FWGrid).

10. REFERENCES

[1] Abilene interior-routing metrics. `http://noc.net.internet2.edu`, March 2006.

[2] D. Andersen, H. Balakrishnan, F. Kaashoek, and R. Morris. Resilient overlay networks. In *Proceedings of the 18th Symposium on Operating Systems Principles*, pages 131–145, 2001.

[3] J. Behrens and J. J. Garcia-Lunes-Aceves. Distributed, scalable routing based on link-state vectors. In *Proceedings of the ACM SIGCOMM Conference*, pages 136–147, 1994.

[4] C. Cheng, R. Riley, S. P. R. Kumar, and J. J. Garcia-Lunes-Aceves. A loop-free extended Bellman-Ford routing protocol without bouncing effect. *ACM SIGCOMM Computer Communication Review*, 19(4):224–236, September 1989.

[5] Cisco Systems. *Introduction to EIGRP*. Document ID 13669.

[6] Cisco Systems. *OSPF Design Guide*. Document ID 7039.

[7] T. H. Clausen and P. Jacquet. RFC 3626: Optimized Link State Routing protocol (OLSR), October 2003.

[8] V. Fayet, D. A. Khotimsky, and T. Przygienda. Hop-by-hop routing with node-dependent topology information. In *Proceedings of The Eighteenth INFOCOM Conference*, pages 79–87, 1999.

[9] D. Fedyk and P. Bottorff. Provider link state bridging (PLSB). IEEE Draft, 2007.

[10] J. J. Garcia-Lunes-Aceves. Loop-free routing using diffusing computations. *Transactions on Networking*, 1(1):130–141, Feb 1993.

[11] F. E. Heart, A. McKenzie, J. M. McQuillan, and D. C. Walden. ARPANET completion report. Technical Report 4799, Bolt, Baranek and Newman, 1978.

[12] P. A. Humblet. Another adaptive distributed shortest path algorithm. *IEEE Transactions on Communications*, 39(6):995–1003, June 1991.

[13] G. Iannaccone, C. Chuah, R. Mortier, S. Bhattacharyya, and C. Diot. Analysis of link failures in an IP backbone. In *Proceedings of the Second Internet Measurement Workshop*, pages 237–242, 2002.

[14] IEEE 802.11s draft standard, 2007.

[15] K. Ishiguro, V. Manral, A. Davey, and A. Lindem. Traffic engineering extensions to OSPF version 3. IETF Draft, 2007.

[16] A. Iwata, C.-C. Chiang, G. Pei, M. Gerla, and T.-W. Chen. Scalable routing strategies for ad hoc wireless networks. *IEEE Journal on Selected Areas in Communication*, 17(8):1369–1379, August 1999.

[17] J. M. Jaffe and F. H. Moss. A responsive distributed routing algorithm for computer networks. *IEEE Transactions on Communications*, COM-30(7):1758–1762, July 1982.

[18] P. Mahadevan, D. Krioukov, K. Fall, and A. Vahdat. Systematic topology analysis and generation using degree correlations. In *Proc. of the 2006 ACM SIGCOMM Conference*, pages 135–146, 2006.

[19] R. Mahajan, N. Spring, D. Wetherall, and T. Anderston. Inferring link weights using end-to-end measurements. In *Proceedings of 2nd Internet Measurement Workshop*, pages 231–236, 2002.

[20] G. Malkin. RFC 2453: RIP version 2, 1998.

[21] J. M. McQuillan, G. Falk, and I. Richer. A review of the development and performance of the ARPANET routing algorithm. *IEEE Transactions on Communications*, COM-26(12):1802–1811, Dec 1978.

[22] J. M. McQuillan, I. Richer, and E. C. Rosen. The new routing algorithm for the ARPANET. *IEEE Transactions on Communications*, 28(5):711–719, May 1980.

[23] P. M. Merlin and A. Segall. A failsafe distributed routing protocol. *IEEE Transactions on Communications*, COM-27(9):1280–1287, September 1979.

[24] J. Moy. RFC 2328: OSPF version 2, 1998.

[25] Y. Ohara, M. Bhatia, N. Osamu, and J. Murai. Route Flapping Effects on OSPF. In *Proceedings of the 2003 Symposium on Applications and the Internet Workshops*, 2003.

[26] V. D. Park and M. S. Corson. A performance comparison of the temporally-ordered routing algorithm and ideal link-state routing. In *Proceedings of the 3rd IEEE Symposium on Computers and Communications*, pages 592–598, 1998.

[27] P. Pillay-Esnault. OSPF Refresh and Flooding Reduction in Stable Topologies. RFC 4136, 2005.

[28] B. Rajagopalan and M. Faiman. A new responsive distributed shortest-path routing algorithm. In *Proceedings of the ACM SIGCOMM Conference*, pages 237–246, 1989.

[29] S. B. Seidman. Network structure and minimum degree. *Social Networks*, 5(3):269–287, September 1983.

[30] A. Shaikh, C. Isett, A. Greenberg, M. Roughan, and J. Gottlieb. A case study of OSPF behavior in a large enterprise network. In *Proceedings of the 2nd Workshop on Internet Measurement*, pages 217–230, 2002.

[31] M. Thorup. OSPF Areas Considered Harmful. Private paper, Apr 2003.

[32] A. Zinin and M. Shand. Flooding Optimizations in Link-state Routing Protocols. IETF Draft, 2000.

APPENDIX

The appendix consists of proofs omitted in the body of the paper.

Lemma 2. *After executing Algorithm 1 the external view T_{uv} satisfies the View Invariant V1 and Constraints S1, S2, and C1.*

Proof. By inspection, for every edge (x, y), $T_{uv}(x, y)$ is assigned either $T_{uv}(x, y)$ or $T^*_{uv}(x, y)$. Therefore, the view invariant holds by Lemma 1.

Now consider the loop in lines 1 through 9; we claim that after it is executed, T_{uv} satisfies Constraints S1 and S2. It is easy to verify that lines 3–5 ensure S1 holds. Also, if $f_u(w) = v$ for some w and (x, y) is an edge in $\pi_u(u, w)$, then $f_u(y) = v$ also. This implies the assignment on line 7 was executed and $e_{uv}(x, y) = e_u(x, y)$ as required.

In lines 10 through 14 the algorithm updates edges to satisfy Constraint C1. We claim that the resulting external view indeed satisfies Constraint C1. First, note that after lines 1 through 9, the distance $d_{uv}(w)$ cannot increase, because $e_{uv}(x, y) \geq e_u(x, y)$ per Constraint S1. Now consider, toward a contradiction, a node w such that $d_{uv}(w) > (1 + \epsilon_u(w))D_u(w)$ and $d_{uv}(w) \neq d_u(w)$. The latter implies that there must be an edge (x, y) in $\pi_u(u, w)$ where $e_u(x, y) < e_{uv}(x, y)$. But then line 12 would have been executed for edge (x, y), and $e_{uv}(x, y) = e_u(x, y)$, a contradiction. □

Lemma 3. *Fix a time $t > \Delta$. If $\phi^t(u, w)$ is a non-empty path that is both quiet during time interval $[t - \Delta, t]$ and coherent at time t, then $\phi^t(u, w)$ is a finite path from u to w and*

$$\|\phi^t(u, w)\|^t \leq d^t_u(w).$$

Proof. Consider the state of the network at the fixed time t. For notational simplicity, we will omit the temporal superscript t. To prove the lemma, we first show that $\phi(u, w)$ is finite, and then show that its last element is w. We then use this fact to prove the bound. We start with two observations.

Observation 1 At time t the path $\phi(u, w)$ has been quiet for duration at least Δ, so the update algorithm has been executed at least once by each node along the path $\phi(u, w)$ during the quiet interval $[t - \Delta, t]$. By Equation 7, $e_x(x, y) = e(x, y)$ for each edge (x, y) in $\phi(u, w)$.

Observation 2 The distance estimate $d_u(w)$ must be finite; otherwise $f_u(w) = \text{NONE}$, implying $\phi(u, w)$ is the empty path.

To show that $\phi(u, w)$ is finite, it is sufficient to show that the estimated distance $d_z(w)$ decreases by an edge cost at each node along the path $\phi(u, w)$. Without loss of generality, consider the first edge $(u, f_u(w))$. Let $v = f_u(w)$ and let $\pi_u(u, w) = uv\alpha$, where α is some sub-path. Then:

$$\begin{aligned}
d_u(w) &= e_u(u, v) + \|v\alpha\|_u \\
&= e(u, v) + \|v\alpha\|_u && \text{by Obs. 1} \\
&= e(u, v) + \|v\alpha\|_{uv} && \text{by Constr. S2} \\
&= e(u, v) + \|v\alpha\|_{vu} && \text{by Coherence} \\
&\geq e(u, v) + \|v\alpha\|_v && \text{by Constr. S1} \\
&\geq e(u, v) + \|\pi_v(v, w)\|_v && \text{by opt. of } \pi_v(v, w) \\
&= e(u, v) + d_v(w). && (\star)
\end{aligned}$$

Thus $\phi(u, w)$ is finite. Now let w' be the last node in $\phi(u, w)$. We claim that $d_{w'}(w) = 0$ and therefore $w' = w$. By Observation 2, $d_{w'}(w) \leq d_u(w) < \infty$. But if $d_{w'}(w) \neq 0$ then by definition $f_{w'}(w) \neq \text{NONE}$, contradicting w' being the last node.

It remains to show that $\|\phi(u, w)\| \leq d_u(w)$. The proof is by induction on the length of $\phi(u, w)$. The base case is length 1 which implies

$$\|\phi(u, w)\| = e(u, w) = e_u(u, w) = d_u(w),$$

as desired. Now consider $\phi(u, w)$ and assume $\|\phi(v, w)\| \leq d_v(w)$ where $v = f_u(w)$. Continuing from $(\star)$,

$$\begin{aligned}
d_u(w) &\geq e(u, v) + d_v(w) \\
&\geq e(u, v) + \|\phi(v, w)\| \\
&= \|\phi(u, w)\|.
\end{aligned}$$

□

Lemma 4. *Fix a time $t > \Delta$. Let β be a path from u to w. If β is (i) quiet during $[t - \Delta, t]$, and (ii) coherent at time t, then*

$$d^t_u(w) \leq (1 + \epsilon)\|\beta\|^t,$$

where $\epsilon = \max_{x \in \beta} \epsilon_x(w)$.

Proof. As in the proof of Lemma 3, consider the state of the network at the fixed time t. For notational simplicity, we will omit the temporal superscript t. Also as in that proof, we claim $e_x(x, y) = e(x, y)$ for each edge (x, y) in β.

The proof of this lemma is by induction on the length of β. If β is the empty path, then $u = w$ and we're done. Now let $\beta = uv\alpha$ for some path $alpha$, and assume $d_v(w) \leq (1 + \epsilon)\|v\alpha\|$. Then, using Coherence in step $(\star)$:

$$\begin{aligned}
d_u(w) &\leq e_u(u, v) + \|\pi_u(v, w)\|_u \\
&= e(u, v) + \|\pi_u(v, w)\|_u \\
&\leq e(u, v) + \|\pi_u(v, w)\|_{uv} \\
&\leq e(u, v) + \|\pi_{uv}(v, w)\|_{uv} \\
&= e(u, v) + \|\pi_{vu}(v, w)\|_{vu} \qquad (\star) \\
&= e(u, v) + d_{vu}(w) \\
&\leq e(u, v) + \max\left\{(1 + \epsilon_v(w))D_v(w),\ d_v(w)\right\} \\
&\leq e(u, v) + \max\left\{(1 + \epsilon)D_v(w),\ d_v(w)\right\} \\
&\leq e(u, v) + \max\left\{(1 + \epsilon)\|v\alpha\|,\ d_v(w)\right\} \\
&\leq e(u, v) + \max\left\{(1 + \epsilon)\|v\alpha\|,\ (1 + \epsilon)\|v\alpha\|\right\} \\
&\leq e(u, v) + (1 + \epsilon)\|v\alpha\| \\
&\leq (1 + \epsilon)\|\beta\|.
\end{aligned}$$

□

Lemma 5. *If a network becomes quiet at some time t, then after a finite period of time it also becomes coherent.*

Proof. Divide the time line after t into epochs of duration Δ. We claim that if none of the views change during an epoch, then they will not change in subsequent epochs and the network is coherent. This is because the Update algorithm is a deterministic function of the views and edge weights, with the property that if the internal view and edge weights do not change, then the current time input is ignored (by Equation 7). Furthermore, from by Equations 6, 7, and 8 it follows that if the external views don't change, then they must be coherent.

Since an edge datum is only injected into the network in Phase I when an edge cost changes, no new edge data are injected after time t. Each view update consists of some number of edge datum values being updated to more recent values from another view. Since there is a fixed number of internal and external views in the network, each view can only be updated finitely many times. It follows that the network can only change a finite number of times after time t. But since the network must change each epoch as shown above, it will stop changing and become coherent in a finite period of time. □