Limiting Path Exploration in BGP*

Jaideep Chandrashekar Zhenhai Duan Zhi-Li Zhang Jeff Krasky

Abstract

Slow convergence in the Internet can be directly attributed to the path exploration phenomenon, inherent in *all* path vector protocols. The root cause for path exploration is the dependency among paths propagated through the network. Addressing this problem in BGP is particularly difficult as the AS paths exchanged between BGP routers are highly summarized. In this paper, we describe why path exploration cannot be countered effectively within the existing BGP framework, and propose a simple, novel mechanism—*forward edge sequence numbers*—to annotate the AS paths with additional "path dependency" information. Then, we develop an enhanced path vector algorithm, *EPIC*, which can be shown to limit path exploration and lead to faster convergence. In contrast to other solutions, ours is shown to be correct on a very general model of Internet topology and BGP operation. Using theoretical analysis and simulations, we demonstrate that EPIC can achieve a dramatic improvement in routing convergence, compared to BGP and other existing solutions.

1 Introduction

The Internet is a collection of independently administered *Autonomous Systems* (ASes), which are glued together by the Border Gateway Protocol (BGP) [RLH03], the *de facto* inter-domain routing protocol in the Internet. BGP is a path vector routing protocol where the list of ASes along the path to a destination (AS path) is carried in the BGP routing messages. Using these "path vectors", BGP can avoid the looping problems associated with traditional distance vector protocols. However, BGP may still take relatively long time to converge following a network failure. Experimental studies show that in practice, BGP can take up to *fifteen* minutes to converge after a failure [LMJ99]. The root cause of this slow convergence is the *dependency* among paths announced through the network, which leads to *path exploration*: when a previously announced path is withdrawn, other paths that *depend* on the withdrawn path (now *invalid*) may still be chosen and announced, only to be removed later one by one. During path exploration, the network as a whole may explore a large number of (valid and invalid) routes before arriving at a stable state. Theoretically, a path vector routing protocol can explore as many as $O(n!)$ alternative routes to converge in the worst case. Addressing path exploration within the framework of BGP is particularly challenging: the AS paths carried with BGP route advertisements are highly summarized, making it difficult to capture dependencies between different paths and to correctly distinguish between valid and invalid paths.

Path exploration has several undesirable side effects. First, it takes several minutes for the network to converge after a failure. In this time, a large number of packets are lost or delayed, adversely

*This paper is to be considered a work in progress.

affecting the performance of applications such as VoIP, streaming video, online gaming, etc. Second, the additional protocol activity increases the load on routers, which are forced to process updates for transient routes. In severe cases, this additional load can cause routers to "tip over", leading to cascaded network failures [CGH02]. Third, normal path exploration may be incorrectly identified as instability (i.e., flapping routes), triggering damping mechanisms at routers [MGVK02]. Lastly, it complicates the task of identifying the root-causes of routing updates, essential in understanding inter-domain routing dynamics [Gri02].

In this paper, we propose a simple and novel mechanism—*forward edge sequence numbers*—to annotate routing updates with path dependency information, so as to effectively address the path exploration problem. Based on this mechanism, we develop an enhanced path vector routing protocol, *EPIC*, which limits path exploration and leads to faster protocol convergence after network failure and repair events. Our solution has the following properties: 1) it considerably improves convergence after a failure. The convergence time following a link/router failure is reduced to $O(D)$, where D is the "diameter" of the Internet AS graph; 2) in contrast to previous solutions which assume a *simplified* setting, our solution is based on a more general and realistic model of BGP operation and AS topology: ASes may contain internal routers and share multiple edges with neighboring ASes; 3) it does not require ASes to expose detailed connectivity information; 4) it can be implemented with fairly modest communication and memory overhead

The remainder of this paper is structured as follows: Sec. 2 briefly reviews BGP operation and illustrates the path exploration problem. In Sec. 3 we introduce the proposed novel mechanism for embedding path dependency, i.e., *forward edge sequence numbers* and use examples to show how they are used. A detailed description of *EPIC* is presented in Sec. 4, along with correctness results. Sec. 7 lists some analytical results for *EPIC* and simulation results are presented in Sec. 8. Finally, we review some related work in Sec. 9 and conclude in Sec. 10.

2 Path Exploration

In this section, we briefly review the operation in BGP and subsequently discuss the path exploration phenomenon. In particular, we show that path exploration is an *inherent* property of *all* path vector protocols and describe why it is particularly hard to address, in the context of BGP.

2.1 Border Gateway Protocol

BGP is used between ASes to exchange network reachability information. Each AS has one or more border routers that connect to routers in neighboring ASes, and possibly a number of internal BGP routers. BGP sessions between routers in neighboring ASes are called eBGP (external BGP) sessions, while those between routers in the same AS are called iBGP (internal BGP) sessions. Note that two adjacent ASes may have more than one eBGP session. We now briefly describe the relevant operation at a BGP router (see [RLH03] for the complete specification).

BGP routers distribute "reachability" information about destinations by sending route updates, containing *announcements* or *withdrawals*, to their neighbors. In the rest of this paper, we implicitly assume a fixed destination, say d.

A route announcement contains a destination and a set of route attributes, including the AS path attribute, which is a sequence of AS numbers that enumerates the different ASes that have been traversed by the route. We denote an AS path as $[A_n A_{n-1} \ldots A_0]$, where A_0 is the *origin AS* to

which d belongs. In contrast, route withdrawals only contain the destination and implicitly tell the receiver to *invalidate* (or remove) the route previously announced by the sender.

When a router receives a route announcement, it first applies a filtering process (using some *import policies*). If accepted, the route is stored in the local routing table. The collection of routes received from *all* neighbors (external and internal) is the set of *candidate routes* (for that destination). Subsequently, the BGP router invokes a *route selection process*—guided by local policies—to select a single "best" route from this set [bgp]. After this, the selected best route is subjected to some *export policies* and then announced to all the router's neighbors. Importantly, prior to being announced to an external neighbor (in a different AS), but not to an internal neighbor, the AS path carried in the announcement is prepended with the ASN of the local AS.

2.2 Network Model

We introduce an abstract model for an idealized (and somewhat simplified) inter-domain path vector protocol that captures and incorporates most of the complexity in BGP. This abstract model frees us from many nitty-gritty details and operational considerations (e.g., policy issues) of BGP, while allowing us to focus on the *relevent* issues.

To represent the two-level inter-domain routing structure of the Internet, we adopt the following two-level graph model. First, we model the Internet as an undirected (border) router-level graph, $\mathcal{G} = \{\mathcal{V}, \mathcal{E}\}$. Here the vertices, $\mathcal{V} = \{v_{il} : 1 \leq i \leq N, 1 \leq l \leq M_i\}$, represent the collection of border routers (e.g., BGP speaking routers): each v_{il}, $1 \leq l \leq M_i$, denotes border router l in AS i, and $\mathcal{V}_i = \{v_{il} : 1 \leq l \leq M_i\}$ is the set of border routers in AS i. Henceforth, we simply use the word router for v_{il}. The edge set $\mathcal{E}$ represents the "logical links" (e.g., BGP Sessions) between the border routers. We distinguish between two types of edges: *internal* edges, i.e., edges between routers within an AS; and *external* edges, i.e., edges between routers in different ASes. For each AS i, we use $\mathcal{E}_i^I$ to denote the set of internal edges in AS i. The collection of external edges is denoted by $\mathcal{E}^X$. Hence $\mathcal{E} = (\cup_{i=1}^{N} \mathcal{E}_i^I) \cup \mathcal{E}^X$.

To simplify our discussion of our abstract path vector protocol model, we impose the following restriction on the set $\mathcal{E}$:

1. $\forall i : \; l \neq l' \Rightarrow \langle v_{il}, v_{il'} \rangle \in \mathcal{E}^I$. In other words, we require that routers within an AS are fully connected, i.e., form a *full mesh*.

However, later in the paper, we discuss situations where this does not hold and how such situations can be handled.

Given this router-level graph $\mathcal{G}$, we can now construct an induced *AS Graph*; $\mathcal{G}_{AS} = \{V_{AS}, E_{AS}\}$. $V_{AS} = \{1, \ldots, N\}$ and $E_{AS} = \{\langle i, i' \rangle : \langle v_{ij}, v_{i'j'} \rangle \in \mathcal{E}^X\}$. We will refer to a vertex in $\mathcal{G}_{AS}$ as an *AS* node, and an edge in $\mathcal{G}_{AS}$ an *AS* edge. (However, when there is no confusion, we will simply refer to them as a node and an edge.) We now describe the abstract path vector protocol. The description below is *with respect to a specific destination* (i.e., a network prefix), say d (in AS 1). Each AS i announces or withdraws network reachability (to destination d) to a neighboring AS based on network reachability it has learned from other AS neighbors. The network reachability is represented in the form of AS-level path vectors, i.e., paths in the AS graph $\mathcal{G}_{AS}$.

Consider a router v_{il} in AS i and let $\mathcal{P}_{il}(d)$ denote the set of *all* AS paths to destination d (in AS 1) that it learns from its neighbors in other ASes. Since d is implicitly specified in our description, we shall simply use $\mathcal{P}_i$ to represent the set of *all* paths at *any* router in AS i. This allows

us to operate at the AS level, rather than considering each router. Abstractly, for each AS i, we define a path preference function Λ_i that selects the "preferred" path, or possibly the empty path, from the set $\mathcal{P}_i$. We define $P_i^* = \Lambda_i(\mathcal{P}_i)$. Since we are discussing *protocol convergence*, we will only consider policy configurations i.e. $\{\Lambda_i : i \in V_{AS}\}$, that are safe (see [VGE99, GW99] for a discussion of this problem). If the path P_i^* happens to be empty (and AS i previously announced a non-empty path), then AS i sends a withdrawal to its neighbors; otherwise it announces the path P_i^*.

We use the term *physical event* to denote a lower-level network event (e.g., a link failure) that causes an edge between two internal or external border routers to be affected (e.g., the E-BGP or I-BGP session between the two BGP routers is down). We use the term (inter-domain) *routing event* to denote the resulting *generation* or *propagation* of routing protocol messages, such as a route withdrawal or a new route announcement. In this respect, we distinguish between *primary* (routing) events and *secondary* (routing) events. A primary (routing) event is generated by the routers that are directly affected by a physical event, and corresponds to the *generation* of a new routing message (route withdrawal or update) in its own right. Secondary (routing) events are triggered in response to a primary event, for example, the generation of a new route announcement upon receipt of a route withdrawal message, or the continued propagation of the route withdrawal message.

This distinction between *primary* and *secondary* behavior is key to *correctly* handling path exploration. In many cases, physical events within the AS, as well as events associated with the external edge, can trigger the same secondary behavior. One of the key ideas that we will describe in this paper is the embedding of additional information that will enable this distinction to be made.

We now define some notation that describes the exchange of messages between nodes. For each AS i, we use $i \rightarrow j{:}A[P]$ to indicate a route announcement[1], namely the announcement of its (current) best path P to its neighbor AS j. Sometimes, to emphasize the two border routers involved, we will use $v_{il} \rightarrow v_{jm}{:}A[P]$ to indicate that the announcement is sent from router v_{il} of AS i to router v_{jm} of AS j. Note that upon receiving $i \rightarrow j{:}A[P]$, AS j will add P into its pathset $\mathcal{P}_j$, replacing the previous best path P' announced by AS i, if it exists.[2] Similarly, we use $i \rightarrow j{:}W[P]$ to indicate a route withdrawal from AS i to AS j, which *invalidates* its previous best path P announced to AS j. Upon receiving $i \rightarrow j{:}W[P]$, AS j will remove P from its pathset $\mathcal{P}_j$.

Note that the receipt of a route announcement $i \rightarrow j{:}A[P]$ or withdrawal $i \rightarrow j{:}W[P]$ *may* trigger AS j to generate a *secondary* event (a route announcement or withdrawal), *depending on whether P will affect the selected best path of AS j.*

In a system as large as the Internet, there is a very high likelihood of many primary events (link failures, router resets, policy changes, and so on) taking place at the same time. In addition, it is possible that many logical edges in the graph share the same underlying physical link, whose failure affects all the logical connections. For ease of exposition in this paper, we will assume that any single time, there is at most a single primary event, and this affects a single edge (which may be internal or external). This allows us to make the description of our solution and the subsequent analysis easy to understand.

[1] To be more general, we use the notation $i \rightarrow \{j_1, j_2, \ldots, j_k\}{:}A[P]$ to indicate that AS i sends the identical announcement to each of $j_1, j_2, \ldots, j_k$, which are neighbors.

[2] More precisely, upon receiving $i \rightarrow j{:}A[P]$, the corresponding border router propagates the router announcement to other routers within AS j, each of them will add P its pathset $\mathcal{P}_{jm}$, replacing the old path P' if it exists.

2.3 Path Exploration and BGP

Vectoring protocols are inherently associated with *path dependencies*: the path selected by a router *depends* on paths learned by its neighbors, which in turn is influenced by the paths selected at the neighbors' peers, and so on. This natural property leads to the so-called *path exploration* phenomenon that prolongs protocol convergence. Note that the *path vectors* that are carried in path vector protocols prevent routing loops, but they cannot avoid path exploration. As a path vector protocol, BGP exhibits path exploration. More significantly, it introduces additional complexity that makes it particularly difficult to address this problem. In the rest of this section, we illustrate the path exploration phenomenon by an example, then describe why, in general, it is *impossible* to avoid it by solely relying on the AS paths associated with BGP routes.

Consider the topology in Fig. 1. Now suppose AS 0 announces a path to destination d. This announcement is received at its neighbors and propagated hop by hop. Finally, when the network converges, AS 5 knows three paths to reach d, i.e. $[3210]$, $[4210]$, and $[7610]$, listed here in the order preferred.

Now consider what happens when the link between AS 0 and AS 1 fails, making d unreachable at AS 1. This failure triggers the following sequence of events: AS 1 sends withdrawals to AS 2 and AS 6. In turn, each of them sends withdrawals to their own neighbors. Eventually, AS 5 will receive withdrawals from each of AS 3, AS 4 and AS 7 (in some order). Suppose the first one was from AS 3; then AS 5 removes the path $[3210]$, selects $[4210]$ as the "best path" and sends it to its (other) neighbors. However, if the withdrawal from AS 4 arrives next, then this "best path" is invalidated and AS 5 selects (and announces) $[7610]$. Finally, after AS 5 receives the withdrawal from AS 7, it invalidates the path announced earlier and *sends a withdrawal*.

This cycle of selecting and propagating (invalid) paths is termed *path exploration*. Clearly, the cycle stops after *all* the obsolete routes have been explored and invalidated.

Path exploration significantly prolongs the protocol convergence after a network failure or repair event. Labovitz *et.al.* showed that in the worst case, as many as $O(n!)$ alternate paths may be explored after a failure [LABJ01]. However, in practice, such a worst-case scenario is rare in todays Internet: common routing policies, which reduce the number of available routes, and protocol timers, that limit how fast updates can be sent, have a beneficial effect. Nonetheless, path exploration can still adversely impact performance, especially after network failures. It is quite common for Internet convergence to take several minutes, and even a relatively short convergence delay can cause pronounced packet loss. This is most severe in the Internet core, where link speeds are very is characterized by high link speeds and rich connectivity. [3]

Having discussed why addressing path exploration is important, we now explain why BGP-specific details make it especially hard to solve this problem, and argue that a new *augmented* mechanism is necessary.

Addressing path exploration in BGP is hard. The crux of this matter is that it is impossible to accurately detect (or even describe) the path dependencies based *solely* on the AS path information carried in BGP announcements. The AS path is a very high level summary of the actual router level paths, and does not reflect the (often) complicated internal AS topologies and interconnections.[4] The devil being in the details, this summarization conceals information that would have made it possible

[3] The so-called tier-1 ISP's peer with each other at more than 15 different locations [coo02].

[4] For instance, large ISPs peer with each other at many locations and the same AS path may be announced at each location. Although these paths correspond to *distinct* routes, this is not reflected in the actual AS paths.

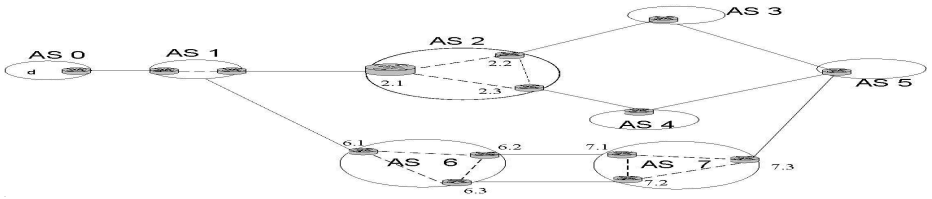


Figure 1: BGP and Path Exploration. Solid lines represent eBGP sessions, while dashed lines indicate iBGP sessions.

to detect path dependencies. In the following, again using Fig. 1, we illustrate how different failure events can generate the same updates, complicating the task of detecting path dependency in BGP.

Note that when the network is in a stable state, AS 5 knows of three routes to d, i.e. $[3210]$, $[4210]$ and $[7610]$. Now, suppose that some external event causes the edge between AS 1 and AS 2 to fail. Then, router 2.1 can no longer reach d. Thus it is the event originator, and generates a withdrawal to invalidate the route $[10]$. This *primary* event will cause both routers 2.2 and 2.3 to withdraw the route(s), $[210]$ previously announced to AS 3 and AS 4 earlier. In other words, this primary event should affect both routes that AS 5 learned earlier through AS 3 and AS 4 (i.e. $[3210]$ and $[4210]$ respectively).

Now consider a different failure event, this time affecting the *internal* edge between 2.1 and 2.2, but *not* the edge $\langle 2.1, 2.3\rangle$. In this case router 2.2 detects the event and is the event originator. Correspondingly, it will generate a withdrawal invalidating the route $[210]$ sent to AS 3 earlier. When the withdrawal is forwarded to AS 5 from AS 3, the only route that should be invalidated is $[3210]$.

In both these distinct scenarios, AS 3 will send a withdrawal to AS 5, (implicitly) invalidating the same $[210]$. How can AS 5 know that in the first case, the withdrawal from AS 3 should cause two routes to be invalidated, and only one in the second case? By simply inspecting the AS path information in the route updates, AS 5 cannot distinguish between these two scenarios.

To make things more complicated, now consider a third scenario, where the internal edge between router 6.1 and router 6.2 fails. This *may* cause router 6.2 (the event originator) to withdraw the route $[610]$ (announced earlier). In turn, router 7.1 will send the withdrawal of $[610]$ to router 7.3. Compare this scenario vs. the scenario where the edge between AS 0 and AS 1 fails, which also causes router 7.1 to send the withdrawal of $[610]$ to 7.3. Can router 7.3 tell that in the former case it can still reach d via router 7.2, but not in the latter? Again, by simply inspecting the AS path information in the route updates, *it cannot*!

Furthermore, multiple network events may occur in a relatively short interval. Due to the general complexity of AS topology and the varied propagation delays along different paths, updates for events may arrive at a router in a different order than which the events occurred. To make this concrete, consider the situation when the edge between AS 0 and AS 1 fails, causing AS 1 to withdraw the previously announced path $[10]$. But now, suppose this is a transient failure, and the edge comes back up quickly; causing AS 1 to re-announce $[10]$. However, the delays along the paths towards AS 5 are different. Then, possibly, the withdrawal and subsequent re-announcement arrive at AS 5 through AS 3 faster than the withdrawal sent along the path $[4210]$. When AS 5 receives this "duplicate" withdrawal from AS 4, it will treat it as a withdrawal for the route $[3210]$, instead of simply discarding it.

These examples clearly illustrate that AS paths, carried with BGP routes, do not contain suffi-

cient information to correctly distinguish valid and invalid paths, which is critical in suppressing the exploration of obsoleted paths. Clearly, to address this problem effectively, we need to incorporate *additional* information into route updates that will: (1) correctly capture the dependencies between invalidated paths, and (2) be able to distinguish between route updates triggered by old and new events. Also, ideally, such a mechanism should not require an AS to expose detailed (or internal) connectivity information, nor impose undue processing, memory or communication overheads upon a router. In the next section we introduce the notion of *forward edge sequence numbers*, which satisfy all these requirements. Using this AS path "annotation", routers can identify *all* routes that are rendered obsolete by some failure event, and invalidate them at once, significantly improving the protocol convergence time.

3 Forward Edge Sequence Numbers

As discussed previously, the AS path route attribute is insufficient to correctly distinguish *invalid* paths from those that are valid.

This is because a router makes no distinction between AS paths it exports to *different neighbors*. In other words, the outgoing (or *forward*) edge is not embedded in the announced AS path. In this section, we describe how the "forward edge" captures the (missing) information that will enable a router to identify paths obsoleted by a failure event. Our solution uses the notion of a *forward edge sequence numbers* (or *fesns*) to capture the "state" of a forward edge. There are two different types of *fesn's* used in our scheme: *major* and *minor*. The former is defined uniquely for a pair of adjacent ASes and is shared across all the minor edges between them. The latter is used to distinguish between routes learned over distinct minor edges (from the neighbor AS).

Formally, a *major fesn* is described as follows: at any AS, say AS X, corresponding to each of its neighbors, say AS Y, we associate a *major fesn* (specific to each destination). We use the notation (X:Y,n) to describe the *major fesn* for the forward edge from AS X to AS Y. Note that the integer n is incremented when $\langle X, Y \rangle$ is restored after a failure. Importantly, it is *not incremented* when the edge fails. Note that (X:Y,n) is "managed" by AS X, i.e., AS X is (solely) responsible for incrementing the sequence number.[5]

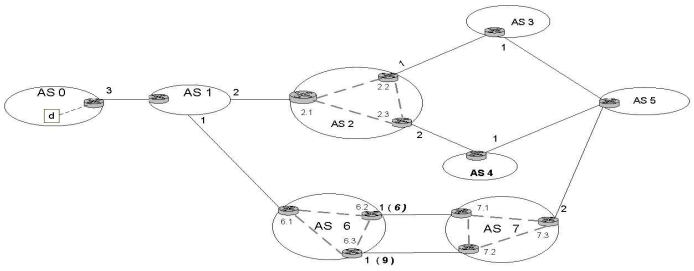
When AS X sends a route announcement to neighbor AS Y, it attaches (more precisely, prepends) the corresponding *major fesn*, i.e., (X:Y,n), to the route.[6] The same operation is performed at every router along the way and consequently, a route contains a list of *major fesn's*, which we call the *fesnList* of the route.

Since *fesns* are distinct, AS X may send the same route update (with the exact same AS path) to neighbors AS Y and AS Z, but the attached *fesn's* are different, i.e., (X:Y,n) and (X:Z,m) respectively. In other words, though the AS paths carried in the route updates are identical, *the corresponding fesnList's are different!* More generally, the *fesnList's* sent to different neighbors are always distinct. This simple property allows us to capture the complex dependencies in AS paths.

If there are multiple minor edges between neighboring ASes, they will *all* be associated with the same *major fesn*. In order to distinguish between routes learned from different routers in the same AS neighbor, a *minor fesn*, specific to each router level peering session, is used. Given router-router

[5]However, in a few special cases, in order to preserve consistency, we may also require AS Y to independently increment the *fesn*.

[6]Hence the designation *"forward edge" sequence number*. When the announcement is from AS X to AS Y, we can consider $\langle X, Y \rangle$ as the forward edge.



AS path	fesnList
3210	{ (3:5,1) (2:3,1) (1:2,2) (0:1,3) }
4210	{ (4:5,1) (2:4,2) (1:2,2) (0:1,3) }
7610	{ (7:5,2) (6:7,1) (1:6,1) (0:1,3) }

Figure 2: AS level topology. Each (internal) router is labelled ASN.router_id. Numbers along edges represent the *fesn* values. The routing table at AS 5 is shown in the table.

edges, say $\langle x', y' \rangle$ and $\langle x'', y'' \rangle$, between AS X and AS Y, we associate them (uniquely) with distinct *minor fesn's*, i.e. $(x'\colon y', k')$ and $(x''\colon y'', k'')$. However, they are both associated with the *same* AS level *major fesn*. Like its counterpart, a *minor fesn* is incremented after the corresponding (minor) edge is repaired. A key difference is that *minor fesns* are only carried in internal routing updates, but never exported to a different AS. For example, in Fig. 2, when 6.2 and 6.3 send announcements over the forward edges to routers 7.1 and 7.2, the corresponding *minor fesn's*, i.e. for $\langle 6.2, 7.1 \rangle$ and $\langle 6.3, 7.2 \rangle$, are attached. These are preserved when 7.1 and 7.2 forward the announcements to internal neighbors. However, this is stripped out from any updates sent to a different AS, for example, when router 7.3 sends a route to AS 5.

In the rest of this section, using examples, we describe how the *fesnlist* is constructed, and how network events i.e., failures and repairs, are handled. A detailed algorithmic description, is presented in the next section.

Consider the topology in Fig. 2; the (major) *fesn's* for each *forward edge* are indicated along the edge (the numbers in parenthesis are the *minor fesn* values). By [ASPATH]{fesnList}, we mean a route announcement containing both the AS path as well as the associated *fesnList*. Thus, the route advertised by AS 0 to AS 1, with ASPATH=[0] and fesnList={(0:1,3)} is written as [0]{(0:1,3)}. When AS 1 propagates this route further to AS 2 and AS 6, the announcements received at routers 2.1 and 6.1 are [10]{(1:2,2) (0:1,3)} and [10]{(1:6,1)(0:1,3)} respectively. Note, between these announcements, the AS path is identical, but the *fesnList's* are distinct.

When routes are advertised internally, the *fesnlist* is not changed. Hence both routers 2.2 and 2.3 will receive the (same) route announcement, i.e. [10]{(1:2,2)(0:1,3)}. When these routers, in turn propagate the route to their neighbors, the announcements received at AS 3 and AS 4 are [210]{(2:3,1)(1:2,2)(0:1,3)} and [210]{(2:4,2)(1:2,2)(0:1,3)}, respectively. Again, notice that the *fesnlist's* are distinct. Finally, when the route announcements have been processed everywhere, the routing table at AS 5 is as shown in Fig. 2.

Following a failure, when an *event originator* generates a route withdrawal, it will insert—into the withrawal—the *fesnList* of the route being withdrawn. When the neighbor (an *event propagator*) generates a subsequent routing update, it sends along the (original) withdrawal *without change.* In other words, an *event propagator* will forward an *exact copy* of the withdrawn *fesnList* it receives. In the case that the *propagator* selects a new *best route* after processing the withdrawal, the *original* withdrawal is "piggybacked" onto the (resulting) route announcement. Thus, every router that receives an update after the failure will see exactly the *fesnList*, inserted by the *originator*. Note that the invalid (or withdrawn) route may not directly correspond to an AS path announced by a router.

To make this distinction clear, we refer to the path described in the *fesnList* as the "path-stem", since all the invalid AS paths are essentially its "branches".

In the rest of this section, we revisit the failure scenarios described in section 2.3 and illustrate how the *fesnlist* attribute can be used to identify the obsolete routes following a failure, avoiding the problems previously mentioned.

External edge $\langle 1, 2 \rangle$ ***fails:*** This is detected by router 2.1 in AS 2, which *originates* a route withdrawal sent to its internal peers 2.2 and 2.3. In keeping with previous notation, a withdrawal message is described as `W:[AS PATH]{fesnList}`. Then the withdrawal sent by 2.1 is W:[10]{(1:2,2) (0:1,3)}.[7] Subsequently, 2.2 and 2.3 will "propagate" the failure event by forwarding the withdrawal to their respective (external) neighbors in AS 3 and AS 4, and they in turn send the withdrawal(s) to AS 5. Note that in each case, the contents of *the withdrawal are identical.*

When it receives a withdrawal message, perhaps attached to a route announcement, a router checks the routes in its routing table and invalidates those that *depend* on the withdrawn "path-stem". In other words, the router invalidates every route for which the *fesnList* attribute contains the withdrawn *fesnList*.

So, when the (first) withdrawal reaches AS 5, the router searches its routing table to identify routes whose *fesnList* contains {(1:2,2) (0:1,3)}. Notice that the first two routes in AS 5's routing table (see Fig. 2) match, and will be removed. However, note that the third route, with AS path [7610] does not satisfy this condition and will be retained. Thus, the *first* withdrawal received at AS 5 will at once invalidate routes learned through AS 3 and AS 4, which both depend on the failed edge. In BGP, each withdrawal message will invalidate a single route, i.e., the route previously announced by the sender.

Now suppose that $\langle 1, 2 \rangle$ is repaired; then AS 1 will increment the *fesn* for the edge. In response to the repair event, it generates a primary routing event, i.e., the route announcement [10]{(1:2,3)(0:1,3)}, and sends it to router 2.1. At 2.1, this *new* route, where the *fesnlist* contains the new value for $\langle 1, 2 \rangle$, is installed in the routing table. The route is then exported to 2.2 and 2.3, which in turn will forward the route to AS 3 and AS 4, and so on. Note that a new route announcement will *always* overwrite an older route from the same neighbor.

Internal edge $\langle 2.1, 2.2 \rangle$ ***fails:*** Notice that this failure will only affect the AS-level path [210] announced earlier by 2.2 to AS 3, but *not* the (same!) AS-level path [210] announced by router 2.3 to AS 4. Since only AS-level paths are announced and withdrawn, router 2.2 will withdraw [210], which it previously announced to AS 3. When AS 3 forwards the withdrawal to AS 5, there are two routes that contain the AS path 210, learned from AS 3 and AS 4. Clearly, this withdrawal should not cause the path learned from AS 4 to be invalidated. This is hard to know without any additional information. This is where the *fesn* becomes important. Note that *fesnList's* announced by 2.2 to AS 3 and AS 4, with the same AS path, i.e., [210], are distinct: the former contains the *fesn* for $\langle 2, 3 \rangle$, while the latter does not (and instead contains the *fesn* for $\langle 2, 4 \rangle$). This ensures that the router at AS 5 can clearly differentiate the same AS path announced by 2.2 to different neighbors.

Thus, in response to $\langle 2.1, 2.2 \rangle$ failing, router 2.2 sends a withdrawal message to AS 3 containing [210]{(2:3,1) (1:2,3)(0:1,3)}, which in turn is forwarded to AS 5. When this reaches AS 5, only the first route (through AS 3) depends on the withdrawn path-stem and is invalidated, while the other two routes are still valid.

[7] As a technical detail, the withdrawal will not explicitly contain the AS Path since it is *embedded* in the *fesnList*. However, we include it here to make the examples easier to follow.

External edge $\langle 6.2, 7.1 \rangle$ ***fails:*** Here, the failure only affects the AS-level path announced by 7.1 to 7.3, but not the route announced by 7.2, associated with the same AS path, i.e., [610]. Thus, when 7.3 receives the withdrawal from 7.1, the route learned from 7.2 should not be invalidated. While the AS path cannot be used to distinguish between the distinct routes, note that that *minor fesn* attribute is different in each route.

Thus, after $\langle 6.2, 7.1 \rangle$ fails, 7.1 (the *originator*) sends withdrawals to 7.2 and 7.3 containing [610]{(6:7,1)(1:6,1)(0:1,3)}*(6.2:7.1,6)*. Note that the *minor fesn* for the failed edge is attached. At 7.3, both *fesnList* and *minor fesn*, as present in the withdrawal, are used to identify invalid routes. Note that only the route learned from 7.1 matches *both* the *fesnList* and *minor fesn*. Thus, 7.3 will invalidate the route from 7.1, but not that from 7.2, which does not match the withdrawn *minor fesn*.

Now, after this, if 7.3 selects the route from 7.2 as its best route, note that the AS path has not changed, and no subsequent route update is required. Alternatively, if it selects a route with a *different* AS path, it will generate a subsequent route announcement.

These examples shows that by embedding forward edge information into route updates, we can correctly detect path dependancies without having to include any information about the internal connectivity in an AS.

4 Detailed Description

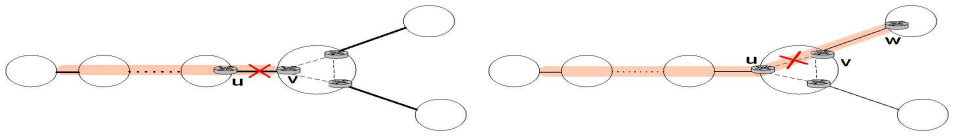
In this section, we present the detailed description of EPIC, which is the enhanced path vector protocol which supports the new route attributes we have defined. As discussed previously, a key notion in our solution is the distinct operations performed by *event originators* and *propagators*. In the following, we separately describe each.

First, we describe some notation to be used in this section. By u^*, we mean the "best route" selected at router u. Recall that when u announces this route to an external neighbor v, it prepends the AS path and *fesnList* with the corresponding values. To make this distinction clear, by $[u \rightarrow v]$, we mean the actual route announced by u to v. Following this notation, we refer to particular attributes associated with each route as $\mathsf{u}^*.\mathsf{attr}$ and $[u \rightarrow v].\mathsf{attr}$, e.g., $\mathsf{u}^*.\mathsf{aspath}$, $[u \rightarrow v].\mathsf{fesnList}$, and so on. In the remainder of this section, we discuss the detailed operation at *originators* and *propagators* in response to various events.

In the description that follows, we make two simplifying assumptions in order to present the main ideas clearly: first, all internal routers in an AS are fully meshed, and second, there is at most a single edge between two adjacent ASes. These restrictions are removed in the longer version of this paper [CDZK04].

Event Originator: A router becomes an *event originator* if a failure or repair is detected on an adjacent edge (or peering session). We first discuss the operation after failure and subsequently the repair scenario. Note that in either case, route updates will be originated *only if the event causes the router to change its best route.*

Failure: When edge $\langle u, v \rangle$ fails, the *originator*, which is node v, does the following: first, it invalidates the route previously learned from u and selects a new best route; then it generates a new routing event if the best route has changed, which is a route withdrawal if no best route exists, or a route announcement is an alternate (valid) route was chosen. Note that in the latter case, a withdrawal is attached to the announcement. In either case, the withdrawal contains the *fesnList* of the



(a) External Failure (b) Internal Failure

Figure 3: The "path stems" that become invalid after a failure are different for internal and external failures. The shaded path segments describe the "path stem" after failure.

invalidated route. Note that a *new* routing event is originated only if the failure affects the best route at v. The detailed algorithm after a failure is presented in Alg. 1. The remaining issue is how the invalid path-stem (or *fesnList*) is identified, and this depends on whether the (failed) edge is internal or external.

First, we consider the case where $\langle u, v\rangle$ is external. This is illustrated using the example of Fig. 3(b). Here, when $\langle u, v\rangle$ fails, all routes that depend on the *shaded* path-stem, which includes $\langle u, v\rangle$, become invalid. Now, this "shaded path-stem" is exactly what is described in the *fesnList* of the route announced by u to v, i.e., $[u{\rightarrow}v]$.fesnList. Importantly, all of v's neighbors previously received the *same* AS path and *fesnList* (in other words, the exact same route). Thus, when an external edge fails, a single (failure) routing event is generated: v will generate a withdrawal containing $[u{\rightarrow}v]$.fesnList and send the same to every neighbor.

The situation is slightly different if $\langle u, v\rangle$ is internal. Here, note that the "edge" $\langle u, v\rangle$ is *not captured in any AS path or fesnList*. However, our notion of "forward edges" can address this easily. The situation is captured in in Fig. 3(a): when the internal edge $\langle u, v\rangle$ fails, the affected path-stem, shown by the shaded area, includes the *forward edge* $\langle v, w\rangle$. Thus, the affected path-stem is described by $[v{\rightarrow}w]$.fesnList, which was exactly what was sent with the earlier route announcement (to w). However, when v has multiple external neighbors, note that the routes announced to each are distinct, as the *fesnLists* are different. Thus, the failure of the internal edge $\langle u, v\rangle$ invalidates each of the (distinct) routes announced by v to its neighbors. To correctly handle the situation, v originates distinct (failure) routing events corresponding to each external neighbor: if w is an external neighbor of v, then v will generate a withdrawal containing $[v{\rightarrow}w]$.fesnList and send it to w. Note that this might be attached to the announcement of an alternate route.

Repair: Recall that *repair* events also trigger route updates and the operation is described in Alg. 2. Here, unlike the failure case, when $\langle u, v\rangle$ is repaired, the identity of the *originator* depends on whether the edge is internal or external. In either case, u will send a route announcement to v, i.e., its best route, but the *originator* is the router that "exports" the event out of the AS.

If $\langle u, v\rangle$ is external, then u is the *event originator*. When it detects the *repair* event, it immediately increments the *fesn* for $\langle u, v\rangle$. Subsequently, it generates a new routing event, i.e., a route announcement, and sends it to v. Note that the *fesnList* carried in the announcement reflects the incremented *fesn*.

Alternatively, if $\langle u, v\rangle$ is internal, then v is the *originator*. If the new route learned from u replaces its current best route, v will *originate* route announcements to send its neighbors. Importantly, v will increment the *fesn* for each of the forward edges prior to sending the announcement. At each of v's neighbors, the route announcement overwrites the older route from v.

Algorithm 1 FAILURE($\langle u, v \rangle$)

@ router v: {notation means "do the following at router u"}
remove $[u \rightarrow v]$ from routing table
v^* = SELECT_BEST_ROUTE( )
if v^* has changed **then**
  ANN = v^*
  if $\langle u, v \rangle$ is external **then** {generate single routing event}
    WDRAW.fesnList = $[u \rightarrow v]$.fesnList
    for all $w \in NEIGHBOR(v)$ **do**
      send (ANN,WDRAW) to w
    end for
  else {generate (distinct) multiple (failure) events}
    for all $w \in NEIGHBOR(v)$ **do**
      WDRAW.fesnList = $[v \rightarrow w]$.fesnList
      send (ANN,WDRAW) to w
    end for
  end if
end if

Algorithm 2 REPAIR($\langle u, v \rangle$)

if $\langle u, v \rangle$ is external **then** {u is the *originator*}
  @ router u:
  increment $fesn(\langle u, v \rangle)$
  ANN = v^*
  send (ANN) to v
else {internal edge repair; v is the *originator*}
  @ router v:
  include $[u \rightarrow v]$ in candidate set
  v^* = SELECT_BEST_ROUTE( )
  if v^* has changed **then**
    ANN = v^*
    for all $w \in NEIGHBOR(v)$ **do**
      increment $fesn(\langle v, w \rangle)$
      ANN = v^*
      send (ANN) to w
    end for
  end if
end if

Event Propagator: When a router receives a route update from a neighbor, it acts as an *event propagator*. In other words, it processes the event captured in the update and may generate a secondary routing update to tell its own neighbors about the event. The route update is processed as follows: if it contains a withdrawal, then all routes in the routing table, which *depend* on the *fesnList* in the withdrawal, are marked invalid; then, any new route announcement is included in the router's candidate set (older routes from the same neighbor are overwritten). Subsequently, a new best route is selected, and if this has changed, then a (secondary) route announcement is generated. On the other hand, if no other routes exist, then the secondary update contains only the (original) withdrawal. Importantly, if a route announcement is generated, then the original withdrawal is piggybacked. The important point here is that a withdrawal received by the propagator, if it causes a route change, is forwarded as is. Thus, every router that learns of the failure event uses the exact same *fesnList*, selected by the *originator*, to invalidate routes.

Clearly, the crucial step in processing a withdrawal is the *correct* identification of routes that *depend* on the withdrawn path-stem. Here, the structure of the *fesnList* attribute makes it easy to identify *dependent* routes. Essentially, if $f_1 = \langle (a_1 \colon b_1, n_1), \ldots (a_k \colon b_k, n_k) \rangle$ is the withdrawn *fesnList* contained in the withdrawal, and $f_2 = \langle (a'_1 \colon b'_1, n'_1), \ldots (a'_l \colon b'_l, n'_l) \rangle$ is associated with some route in the candidate set, with $k \leq l$, we can say that the f_2 depends on the withdrawn path-stem f_1 if and only if

$$(a_i = a'_i) \wedge (b_i = b'_i) \wedge (n_i \geq n_i) \quad i = 1, \ldots, k$$

Note that the comparison is performed *fesn* by *fesn*. Two *fesns* are comparable if they correspond to the same edge. The last conjunction follows from the fact that the *fesn* for an edge is incremented every time it is repaired; hence larger *fesn* values indicate "newer" information. Thus, any routes that "match" in the manner described are invalid and will be *excluded* from the set of candidate routes. It should be stressed that if a received withdrawal causes a route change, then the exact same withdrawal is sent with any route updates to all the router's neighbors.

When either of the assumptions made at the start of this section fail to hold, there is a "loss of visibility" within an AS. For example, *route reflection* used in very large ISP networks, in which case the internal routers are not fully meshed, and a particular router might not know the route selections at others. This "loss of visibility" introduces additional complexity into BGP and we need additional mechanisms to address them. For instance, we can correctly handle multiple edges between ASes by using *minor fesns*. In later sections we detail how our solution can handle such situations.

4.1 Correctness

Here, we establish some theoretical properties of our solution. In particular, we show that following a single failure (routing) event, no router will select an invalid route.

Lemma 1. *At an event originator, upon a failure event, the withdrawal contains the invalid path stem.*

Proof. When $\langle u, v \rangle$ fails, all routes that depend on $\langle u, v \rangle$ are invalid. In other words, any route that was announced from u to v are invalid. When the *external* $\langle u, v \rangle$ fails, the withdrawal contains $[u \rightarrow v].\mathsf{fesnList}$. This explicitly contains the edge $\langle u, v \rangle$, and the result follows.

Now, if $\langle u, v\rangle$ is internal, then $[u{\rightarrow}v]$.fesnList does not contain the failed edge. However, if w is any (external) neighbor of v, the withdrawal sent to w contains $[v{\rightarrow}w]$.fesnList. Note that the route $[v{\rightarrow}w]$ does not depend on any other internal edge in the same AS as v. Thus, when $\langle u, v\rangle$ fails, $[v{\rightarrow}w]$.is rendered invalid, and the result follows. □

Lemma 2. *After a withdrawal is processed at an* event propagator, all *routes that depend on the invalided path are removed from the routing table. Also, no valid route is removed.*

Proof. It is trivial to show that *all* invalid routes are removed when a withdrawal is processed.

Now suppose that a *valid* route, say v, associated with an *fesnList*, $v_f = \langle(a'_1{:}\, b'_1, n'_1), \ldots, (a'_k{:}\, b'_k, n'_k)\rangle$ is invalidated when a withdrawal, w, containing $w_f = \langle(a_1{:}\, b_1, n_1), \ldots, (a_k{:}\, b_k, n_k)\rangle$ is received. Clearly, $k \leq l$ and $\langle a_i, b_i\rangle = \langle a'_i, b'_i\rangle$, for $i = 1, \ldots, k$ as otherwise, v would not have been invalidated.

Now, if the withdrawal was originated due an external (failure) event, then $\langle a_k, b_k\rangle$ must have failed and v *cannot be valid*.

On the other hand, suppose that the withdrawal was due to an internal event, in which case, $\langle b_{k-1}, a_k\rangle$ has failed. Since the route v was invalidated, we have $\langle a'_{k-1}, b'_{k-1}\rangle = \langle a_{k-1}, b_{k-1}\rangle$ and $\langle a'_k, b'_k\rangle = \langle a_k, b_k\rangle$

It follows that v was announced over the failed edge $\langle b_{k-1}, a_k\rangle$, and cannot be a valid route after the failure. □

Theorem 3. *In EPIC, following a single failure event, no router selects (and propagates) an invalid path.*

Proof. Suppose that the edge $\langle u, v\rangle$ fails and a single routing event is originated. Without loss of generality, we can assume that the failure affects the best route at v (the event originator), and forces router v to send a route update. Otherwise, no new routing event is generated.

First, note that the event originator will not announce a route that depends on the failed edge (from lemma 1). Suppose that router s is the *earliest* router that, in response to a route update, announces an invalid route that depends on the failed edge. We show that this leads to a contradiction.

Since the failed edge affects the best route at v, the new route update sent by to its neighbors must either be a withdrawal or a new route announcement piggy-backed with a withdrawal (containing the appropriate *fesnList*). Then, if the withdrawal reaches s, which is an *event propagator*, it will *invalidate* any existing route that depends on the failed edge (by lemma 2). Hence route s *cannot* possibly announce a new route update that depends on the failed edge. The only scenario in which this could happen is if *none of the route updates received at s contain a withdrawal.*

Now consider the invalid route announced by router s. Since it depends on the failed edge, we must have a "propagation chain", $s, i_n, i_{n-1}, \ldots, i_1, vP$, where P is the previous best route at v (now being withdrawn), and i_j, $j = 1, \ldots, n$, are *propagators* that "forwarded" the route updates to s. Since v generates a withdrawal, all the nodes in the chain $i_1, \ldots, i_n$ will propagate the withdrawal. Thus s will receive the withdrawal originated at v. This contradicts the earlier statement that none of the route updates received at s contains a withdrawal. □

	Core Router	Campus Router
Prefixes	128735	123632
Routes	5284198	393230
Number of AS paths	731853	64287
Memory used for AS paths	6877.13 kB	788.54 kB
Memory for `fesnList`	14872.44 kB	1652.59 kB

4.2 Overhead

Here, we briefly discuss the additional overhead introduced by *EPIC*, in memory and communication.

Recall that route announcements carry an additional *fesnList* attribute which will be stored in a routers' routing table. In the following, we estimate the additional memory required at a router. For any given route, the length of the *fesnList* is at most the AS path length. Thus, any *fesnList* contains at most L *fesns*, where L is the longest AS (policy permitted) path length. Also, a router will receive at most one route (to a destination) from each of its neighbors. Thus, a router with degree d, will have to store $O(dLP)$ *fesns*, where P is the the number of routed prefixes in the Internet. In practice however, most AS paths in the Internet tend to be much shorter than L, and furthermore route aggregation tends to reduce number of prefixes, so we expect the actual overhead to be lower.

Also, note that the *fesn* does not need to explicitly describe the edge it is associated with. Correct operation only requires that *fesn's* be distinct, i.e., the *fesn* $\langle X, Y \rangle$ is unique. Thus, it is possible to use a "compressed" representation of each *fesn*. For example, using $X(\mathsf{xor})Y$ to "encode" the edge, and 2 bytes for the sequence number, each *fesn* can be described with only 4 bytes. Table 4.2 lists the estimated memory requirements to store the *fesnList* attribute. Note that data for the "core router" was obtained from Route-Views [Vie00].

In addition to storing the *fesnList* attribute in the routing table, a router needs to keep track of *major fesns* associated with adjacent "forward edges", i.e., *external* BGP sessions. Since only the *fesn* values are stored, an additional $O(2dP)$ bytes of memory is required.

With respect to the communication overhead, first note that EPIC will not generate more routing updates than BGP. Also, every *EPIC* routing announcement and withdrawal carry the additional *fesnList* attribute, which slightly increases the size. However, the benefit of using this additional information is that invalid paths are not explored and there is less protocol traffic. Thus, the cost of carrying the additional information is offset by the reduced protocol traffic during convergence. However, we do not explicitly investigate this tradeoff in this paper.

5 Operation with multiple edges

In this section, we extend our scheme to handle instances where there are multiple peering sessions between ASes. However, as in the previous section, we assume that internal routers are fully meshed. This leads to the property that each router knows the best route selected at all the other internal routers. We later discuss special circumstances where the internal "full-mesh" assumption does not hold.

Recall that the *fesn* is defined for a (logical) AS-AS edge. If there are multiple edges (peering sessions) between neighboring ASes, they will *all* be associated with the same *fesn*. In order to

distinguish between routes exchanged over the different edges, an additional tag, namely the *minor fesn*, specific to each router level edge, is used. Given router level edges, say $\langle X.u_1, Y.v_1\rangle$ and $\langle X.u_2, Y.v_2\rangle$ between AS X and AS Y, we associate them (uniquely) with distinct *minor fesn's*, i.e. *(X.u_1:Y.v_1,k_1)* and *(X.u_2:Y.v_2,k_2)* respectively, k_1 and k_2 being the particular sequence number values. However, note that they are *both associated with the same AS level* fesn *(X:Y,n).* To make the distinction clear, we henceforth refer to the AS level "tags" as *major fesn's*.[8] Clearly, the *major fesn* is only incremented after *all* the edges between the ASes have failed, and one of them is subsequently repaired. An important distinction between the two types of *fesn* is that the *minor fesn* is carried in route updates sent internally, but never exported to a different neighboring AS. Thus, the scope of the *minor fesn* is limited to the ASes that share the edge.

In the rest of this section, using the topology of Fig. 2, we briefly how to correctly handle failure and repair events. Notice that in Fig. 2, there are *two* minor edges between AS 6 and AS 7. To simplify the description, we denote the route learned at over $\langle 6.2, 7.1\rangle$ (and $\langle 6.3, 7.2\rangle$) as p_1 (and p_2). Due to a lack of space, we will only discuss the operation for *event originators*. The operation at an *event propogator* is largely similar to that described to the previous section.[9] We discuss failure events first and subsequently, repairs. Note that in either event, an *event originator* will generate a route update *only if the event causes a change in the best route.*

First, we consider the failure scenario. When a particular *external* edge between ASes fails, and a routing update is generated, the information contained in the update is different depending on whether the *originator* knows of other routes learned over other existent edges *between the same ASes*. To start with, we discuss the case where the originator does not know any such routes. To illustrate, consider that $\langle 6.2, 7.1\rangle$ fails and that the routing table at 7.1 does not contain p_2. This might happen if p_2 is not the *preferred route* at 7.2 or because p_2 was withdrawn previously. Note that in either case, no other router in the same AS (as 7.1) is using a route that depends on p_1.[10]. In this case, 7.1 can behave as if *all* the edges between AS 6 and AS 7 are down and simply invalidate *all* routes that depend on the AS-AS edge $\langle 6, 7\rangle$. In other words, 7.1 will generate a withdrawal with $[6.2{\rightarrow}7.1]$.fesnList. Note that the *minor fesn* is not included, as that would cause the *more specific route*, dependent on $\langle 6.2, 7.1\rangle$. but not $\langle 6.3, 7.2\rangle$ to be invalidated.

The alternate situation is that $\langle 6.2, 7.1\rangle$ fails, but router 7.1's routing table contains p_2. Notice that p_2 was first learned by 7.2 over a different minor edge, i.e. $\langle 6.3, 7.2\rangle$, and then re-announced to 7.1. This means that there is at least one router, i.e. 7.2, whose best route (p_2) depends on the (valid) edge $\langle 6.3, 7.2\rangle$. Hence 7.1 *cannot invalidate the edge between* AS 6 *and* AS 7. To ensure correctness, router 7.1 does not invalidate any routes. Instead, it simply originates a route announcement for its new best route. The resulting route does not have an accompanying withdrawal. Note that 7.1 knows of at least one other route– the route announced by 7.2– otherwise the first scenario will apply. Hence a route announcement is always generated in this particular case.

In the rest of this section, we discuss how a repair event is handled. Unlike the previous section, there are now two types of *fesn's —major* and *minor*—that need to be updated. In particular, note that the *major fesn* requires synchronization across all the routers that shares edges. Suppose that the *external* edge $\langle u, v\rangle$ is repaired. Unlike in the previous section, we now consider v to be the event

[8] We could think of the *major fesn* as being associated with a *virtual edge* between ASes. This *virtual edge* is "up" as long as at least one minor edge is "up", and is considered "down" when *all* of the minor edges have failed.

[9] The only additional detail is that *minor fesn* also needs to be matched while invalidating routes.

[10] This follows because all routers are fully meshed. Hence if 7.1 does not know of a route from 7.2, no other router will.

originator. Note that there is no problem updating the *minor fesn* and the *originator* will always do it after it detects the *repair*. On the other hand, since the *major fesn* is "shared" by multiple minor edges an individual *event originator* cannot unilaterally increment the *major fesn*.

Analogous to the failure scenario, we again have to consider two possibilities at the *originator* when an *external* edge is repaired: (1) there are other routes (sharing the same *major fesn*, but learned over other edges) known to the event originator, and (2) no other routes share the same *major fesn*.

Note that the first situation is trivial to handle; since there are other (valid) minor edges at the time of repair, the *major fesn* will not be affected (recall that the *major fesn* changes only after *all* minor edges fail and one of them is subsequently repaired). However, if none are found—the second situation applies—the *originator* cannot assume anything about the other shared edges; they may all be invalid or alternatively, some of them are valid, but the respective ingress routers do not prefer the routes learned over them. To illustrate this, consider Fig. 1 and suppose that $\langle 6.2, 7.1 \rangle$ has just been repaired. Also, assume that 7.2 is not using p_2 as the best route. Note that p_2 might exist in router 7.2's routing table. Hence, 7.1 cannot assume that *all* other edges between $\mathrm{AS}\,6$ and $\mathrm{AS}\,7$ are down (or invalid).

Now, suppose the *originator* were to *unilaterally* increment the *major fesn*, then to ensure correctness, it must *immediately* communicate this to all the other internal routers sharing edges. Rather than impose a separate out of band mechanism to achieve this, we can simply utilize the following fact: if the originators' best path changes after the repair, it will send route announcements to its neighbors. Based on this observation, we can correctly enforce synchronization as follows.

If the route learned over the *repaired* edge is selected as the *best route*, then increment the *major fesn* and send out a route announcement. However, if this is (new route) *not* the best route, then **flag** the *major fesn* and conditionally increment it. At any time, if a route is announced to some neighbors and a flag is associated with a *major fesn* in the route, then clear the flag prior to sending the announcement. Alternatively, if the router receives a route announcement from an internal neighbor, where the *fesnList* contains the *flagged major fesn*, update the routers' instance of the *major fesn* with the new value and reset the flag.

In case of an internal edge being repaired, the situation is identical to the previous section and we omit it here. In the next section, we establish the "correctness" properties of our scheme and show that a router will never select or propagate an invalid route.

6 Route Reflection and Policy

In our description so far, we implicitly assumed that the iBGP connections in an AS were fully meshed, i.e. each internal router maintained iBGP sessions with every other internal router. However, in few large ISP's (with several hundred routers), *route reflection* is used instead.

The introduction of a hierarchy within an AS defeats the assumption made in the previous section —"a router will know when *all* minor edges fail". With route reflectors in place, a border router can no longer be sure to know of *all* the routes available to the AS. Along another dimension, routing policies configured within an AS can also have the same effect. For example, a router could be configured not to re-advertise a particular route to specific internal neighbors. In both of these cases, internal routers no longer have a complete view and may possibly not know of *all* the minor edges, or be notified when they fail. To illustrate this, consider the topology in Fig. 4. Here, there are multiple edges between $\mathrm{AS}\,\mathrm{X}$ and $\mathrm{AS}\,\mathrm{Y}$, i.e. $\langle x_1, z_1 \rangle$ and $\langle x_2, z_3 \rangle$. Suppose the ranking of

routes at $RR1$ is as indicated along the edges. Now, when $\langle u, v \rangle$ fails, $RR1$ selects the alternate path through router z_3 and sends it along to $RR2$ and $RR3$ along with a withdrawal for the path stem corresponding to $[x_1 \rightarrow z_1]$, which eventually reaches router z_7. Lets say that $\langle x_2, z_2 \rangle$ fails soon after. Since this does not affect the current best route at $RR1$, i.e. the route learned from x_3, *no withdrawal is triggered*, and consequently z_7 cannot really know that *all* minor edges between AS X and AS Z have failed. Note from the previous section that *if all minor edges have failed*, z_7 would have originated a withdrawal for the path stem $(0 \ldots X)$

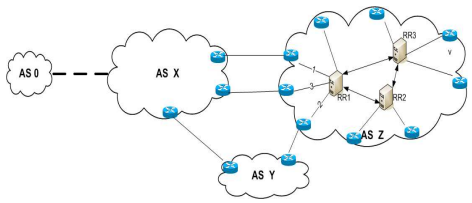


Figure 4: Internal Hierarchy and information abstraction. The labels on edges adjacent to $RR1$ indicate the preference. The RR's are connected in a full mesh and the router v is a client of $RR3$.

In spite of this additional complexity, our separation of roles between *event originator* and *event propagator* enables our solution to handle such situations. Suppose we assume that the routers within an AS do not have *complete knowledge* of the topology and routing policy (at other routers). This can be easily indicated by a flag that is set consistently at each router in the AS. Now, when a border router receives a withdrawal for an external edge, for example when z_7 receives a withdrawal for $\langle x_1, z_1 \rangle$, it originates a *different* event that mimics an internal edge failure. In other words, z_7 will generate a withdrawal that includes the *forward edge*. Stated differently, an egress router will originate a *different* routing event, for the failure of an internal edge, when it receives a withdrawal corresponding to an external edge failure. Note that *any* external failure will lead to the same behavior, even when the external edge is the only one between the neighbors. This follows directly from the router being unaware of other edges.

This "loss of information" does inflict a penalty in terms of performance since we cannot be aggressive—as before—while invalidating routes. However, we can still *guarantee* correct operation by taking a slightly conservative approach. Moreover, we estimate that less than 100 of the 15,000+ ASes actually employ Route Reflection. So only in cases where a failure is local to one of these, EPIC performance will be slightly reduced (though no worse than BGP).

7 Performance Analysis

Table 1: Performance bounds for fail-down.

	BGP	Ghost Flushing	EPIC
Time	$(L-2)\Delta$	$(L-2)h$	$(D-1)h$
Message	$(L-2)(\|E\|-1)$	$\frac{2h(L-2)(\|E\|-1)}{\Delta}$	$\|E\|-1$

Table 2: Performance bounds for fail-over.

	BGP	Ghost Flushing	EPIC
Time	$(\hat{L}+\hat{D}-1)\Delta$	$(\hat{L}-1)h+\hat{D}\Delta$	$\hat{D}(h+\Delta)$
Message	$(\hat{L}+\hat{D}-1)(\|E\|-1)$	$2(\|\hat{E}\|-1)(\hat{D}+\frac{(\hat{L}-1)h}{\Delta})$	$(\|\hat{E}\|-1)\frac{\hat{D}(h+\Delta)}{\Delta}$

Here, we derive upper bounds for the time and communication complexity of EPIC, and contrast it with the performance of BGP and Ghost Flushing[BBAS03].

G	The abstract AS level graph (V, E)
V	The set of AS nodes.
E	The set of AS-AS edges.
N	Size of the graph i.e. $N = \|V\|$
P_s	The "best path" selected at node s, with length $\|P_s\|$
D	Diameter of G i.e. $D = \max_{s \in V} \{\|P_s\|\}$
L	Length of a longest simple path in G
Δ	Hold timer for path announcements
h	Processing delay at a node

Table 3: Notation used in the analysis.

In order to make the analysis tractable, we use the discrete-time synchronized model described in [LABJ01]: in each time step, a node processes *all* of the messages received in the last stage and then selects a single "best path". Then, if the best path has changed, it is exported to all its neighbors. Withdrawal messages are processed differently in each: in BGP, withdrawals are sent only if a node has no valid paths to the destination; Ghost Flushing (GF) sends a withdrawal message when a node receives a route announcement for a longer path, even if alternate routes exist in the routing table. Finally, in EPIC, a node forwards a withdrawal message *only* if the received withdrawal causes the best path to change.

We analyze the bounds in two distinct failure scenarios: *fail-down and fail-over*. A *fail-down* event is one which causes the network to become partitioned. In other words, after the event, some destinations become unreachable, i.e., there is no valid path to reach them. In a *fail-over* situation, there is some valid alternate path available and the network is still connected after the failure. The notation that will be used in the derivations is enumerated in Table 3.

7.1 Fail-down case

The upper bounds in the fail-down scenario are summarized in Table 1. The performance bounds for BGP and Ghost Flushing follow directly from the results in [LAWS01, BBAS03], so we only derive the bounds for EPIC here.

Proposition 4. *After a fail-down event, the convergence time in EPIC is at most* $(D-1)h$*. The number of messages generated during convergence is bounded by* $(|E|-1)$

Proof. When a withdrawal message is received at a router, *all* paths that depend on the failed path stem are invalidated. Clearly, in a fail-down situation, there are no alternate paths (the network is

disconnected) and *every* path in the routing table will be invalidated. Since it has no valid path, the router will send a withdrawal to all of its neighbors.

Note that the node farthest from the location of the failure is at most $(D-1)$ hops away. Since each node takes up to h to process the and forward withdrawal, all nodes receive a copy of the the withdrawal within $(D-1)h$ time after the first withdrawal is generated.

To estimate the number of (withdrawal) messages, notice that at most one withdrawal message will be sent across any edge after the failure. This follows from the following fact: when a router processes the first withdrawal after the (fail-down) event, *all* the paths are invalidated and a withdrawal is sent. Subsequent withdrawals received by the router have no effect and are discarded. Moreover, no message goes across the failed edge. Therefore the number of messages generated in the network is at most $(|E|-1)$. □

7.2 Fail-over case

In this section, we analyze the performance of BGP, Ghost Flushing, and EPIC in the fail-over case. The analysis here is a little more involved and we introduce some additional notation.

First, assume some fixed destination. When a link fails, and alternate paths (to the destination) exist, we can partition the set of nodes into $\hat{V}$ and $V - \hat{V}$ as follows: if the best path at a node is invalidated after the failure, it is in $\hat{V}$. Also, all of this node's neighbors are in $\hat{V}$.

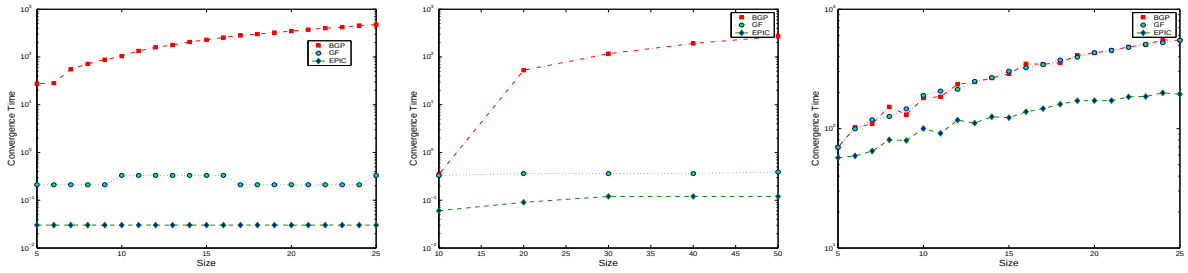
Informally, $\hat{V}$ contains nodes whose *preferred path* is affected by the failure, along with their direct neighbors. Clearly, the nodes in $V - \hat{V}$ are not affected by the failure and will not take part in the convergence process i.e., they will neither receive nor propagate any updates, since their best path does not change after the failure. Thus, we can imagine a "zone of convergence" in G, defined by the induced graph $G[\hat{V}] = (\hat{V}, \hat{E})$. Finally, the last piece of notation we will use is $\hat{L}$, which is the length of *any* longest path from a node in $\hat{V}$ to the destination i.e. $\hat{L} = \max_{v \in \hat{V}} \{|P_v|\}$. Note that the path with length $\hat{L}$ is not constrained to lie in $G[\hat{V}]$. To simplify the analysis we assume shorter paths are preferred.

The performance bounds of the protocols in the *fail-down* case are summarized in Table 2 and are derived below.

Proof. To begin with, lets assume that the nodes in $\hat{V}$ do *not* have a valid, alternate path to the destination. From the previous result for BGP (in the fail-down case), we know that it takes at most $(\hat{L}-1)$ for all these nodes (in $\hat{V}$) to realize that there is no other path. We could call this the *flush out* phase, where the "stale" information about the *invalid* paths is removed from the system.

After this, suppose that the nodes in $\hat{V}$ start receiving updates for the valid paths from the boundary nodes in the "zone of convergence", then it takes at most $\Delta\hat{D}$, for all the nodes to learn about the alternate paths. We could call this stage as the *re-convergence* phase. The subtlety in the proof is that these two different stages can overlap —when an node first learns about the alternate path that it will eventually use (after convergence), there might be other *invalid* paths in the routing table. Consequently, the valid path might not be chosen until *all* the invalid paths have been removed.

Assume the network failure occurred at time t_0. Then suppose $i \in \hat{V}$ learns of its (eventual) best path P_i^* at t_1 ($t_1 > t_0$). Note that P_i^* might not be selected for a while, but it will be the i's best path when the network converges. We know that if there are no other *invalid* paths in the routing table at i, P_i^* will be selected and propagated. In other words, the selection of P_i^* could be delayed till the *flush out* stage is complete. Since the length of the *flush out* stage is at most $(\hat{L}-1)\Delta$,



(a) Clique Fail-down (b) Waxman Fail-down (c) Clique Fail-over

Figure 5: Convergence Time in Clique and Waxman topologies. In all the graphs, the bottom curve represents the performance for EPIC.

clearly the system has no *invalid paths* at $t_0 + (\hat{L} - 1)\Delta$. It follows that P_i^* will be selected and propagated no later than $t_0 + (\hat{L} - 1)\Delta$. Since the diameter of the affected region is $\hat{D}$, it takes at most $\hat{D}\Delta$ for the alternate path(s) to be available at all the nodes in $\hat{V}$. In other words, at time $t_0 + (\hat{L} - 1)\Delta + \hat{D}\Delta$, all the nodes would have converged. Therefore, the convergence time is bounded above by $(\hat{L} + \hat{D} - 1)\Delta$.

To derive message complexity, note that in each interval Δ, at most one message is sent from a node to its neighbor i.e. the rate to sending messages is $1/\Delta$. So in the time taken to converge, at most $\hat{L} + \hat{D} - 1$ messages are sent across an edge. Thus, the number of messages generated is bounded by $(\hat{L} + \hat{D} - 1)(|\hat{E}| - 1)$ □

Proposition 5. *After a* fail-over *event, EPIC will converge within* $\hat{D}(h + \Delta)$ *and require at most* $(\hat{D}(h + \Delta)/\Delta)(|\hat{E}| - 1)$ *messages.*

sketch. From the description of the protocol in Sec. 4, note that a router will does not select and propagate a path unless the current "best path" changes. Now consider all the nodes whose best path is invalidated by the link failure. When these nodes receive a withdrawal (perhaps attached to an announcement), they will each select a new best path and send a route announcement to their other neighbors (with the same withdrawal attached).

Clearly, the nodes that send and/or receive route announcements are exactly the nodes in $\hat{V}$. Thus, if $\hat{D}$ is the diameter of the induced graph $G[\hat{V}]$, it takes $\hat{D}h$ for the *withdrawal* information to reach all the nodes in $\hat{V}$. On the other hand, in the reconvergence process, a node might wait up to Δ time before announcing a new path to its neighbors, and thus the delay on the longest path is exactly $\hat{D}(h + \Delta)$. Since the network does not converge until the nodes learn of the alternate paths, the convergence time is dominated by $\hat{D}(h + \Delta)$, and the result follows.

To derive the message complexity, note that in each interval Δ, at most one message is sent from a node to its neighbor. In other words, the rate of sending messages is $1/\Delta$. So, in the time taken to converge, at most $\hat{D}/\Delta(h + \Delta)$ messages are sent across *each edge*, and the message complexity is bounded by $(\hat{D}(h + \Delta)/\Delta)(|\hat{E}| - 1)$. □

8 Simulation

In this section, we discuss the results of simulation results carried out with the SSFNet simulation package. In particular, we contrast the performance of our solution against BGP and GhostFlushing [BBAS03].

We used different topology families for our simulations: Cliques, Waxman Random Graphs ($\alpha = 0.3, \beta = 0.4$), and Barabasi-Albert (BA) Random Graphs. The latter two topology families were generated using the Brite topology generator [bri]. In each topology, links were assigned a uniform propagation delay of 300ms. Furthermore, for each protocol, we used an `MRAI` value of 30 seconds, which is the BGP default.

In each topology, we simulated both *fail-down* and *fail-over* scenarios. We discuss two performance metrics: convergence time and message complexity. Due to lack of space, we only include results for the clique and Waxman topologies. In the following, we describe the fail-down and fail-over characteristics.

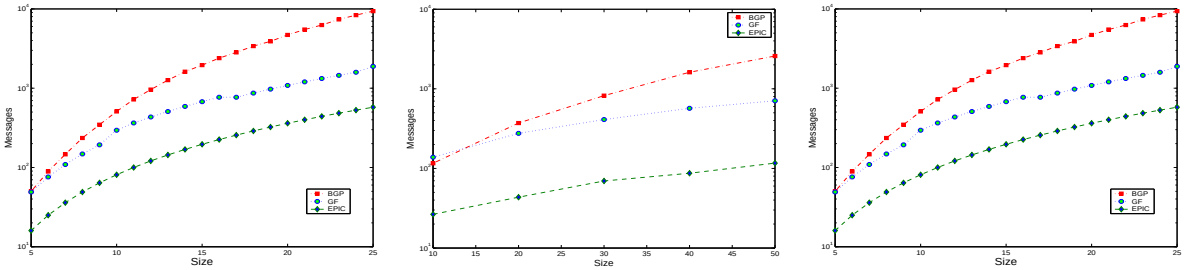
8.1 Results for fail-down

In this scenario, in each simulated experiment, a dummy node is attached to a single node in the network and is disconnected by the failure event. In the clique(s), the additional node is attached to node 0, while in the other topology families, we repeat the simulations varying the attachment points over *all* the other nodes. Simulation were repeated multiple times with different random seeds, and the average performance is plotted.

In Figs. 5(a) and 5(b), we plot the network size against the convergence time, with the y-axis shown in logscale. In each of these, the values for EPIC, which is the bottom line, are far better Ghost Flushing or BGP (which is the topmost curve in each graph). Notice that in the clique topology, i.e., Fig. 5(a), the convergence time for EPIC is constant. This can be explained as follows: all the nodes are directly connected to node 0, which originates the withdrawal event. In EPIC, the first withdrawal will cause a node to invalidate *all* existing routes (since every path contains the failed edge), and the network converges immediately. The constant value corresponding to the EPIC curve is the link propagation and processing delay. On the other hand, BGP and Ghost Flushing cause the alternate (invalid) paths to be flushed from the system one by one, though at different rates. Note that Ghost Flushing performs better than BGP, as it is aggressive in generating withdrawals. However, it performs worse than EPIC because each withdrawal only invalidates the previously announced route, while in the case of EPIC, *all the routes* are invalidated.

In Figs. 6(a) and 6(b), we plot network size against the number of messages generated during the convergence period. We see that EPIC generates far fewer messages than the other two protocols. In Fig. 6(a), i.e., the clique graphs, EPIC causes approximately $(n-1)^2$ withdrawals to be generated: every node other than 0 receives a withdrawal from 0 and subsequently forwards the withdrawal to every other node (the previously announced path must be invalidated).

Unlike the clique topologies, it is difficult to analytically estimate the convergence time or the message volume in the Waxman and BA graphs, since the different sized graphs are independently generated. Nevertheless, the graphs plotted in Figs. 5(b) and 6(b) clearly illustrate that EPIC generally performs better than BGP and Ghost Flushing. This is as expected, since when the dummy node is disconnected, there are no valid alternate paths, and EPIC causes only withdrawals to be generated. In the other protocols, we expect a certain number of invalid paths to be explored. Though



(a) Clique Fail-down (b) Waxman Fail-down (c) Clique Fail-over

Figure 6: Number of Messages generated during convergence. Note that 6(a) and 6(b) correspond to *fail-down*. Again, in all the plots, the bottom curve represents EPIC performance.

not included here, we found the results for the BA topologies to be qualitatively similar to what we described for the Waxman graphs.

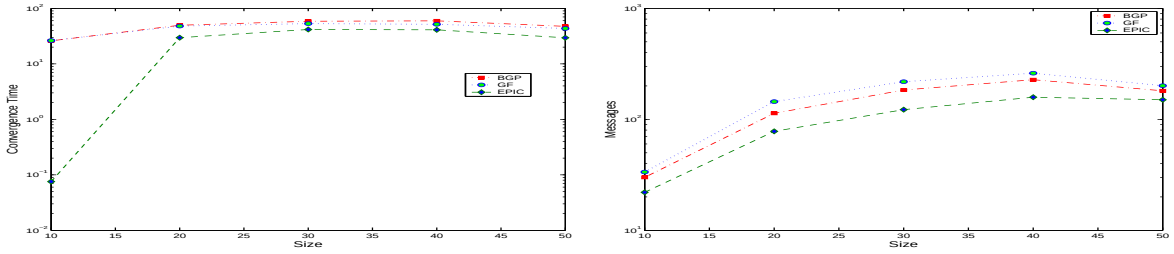
8.2 Results for fail-over

In this scenario, a *dummy node* is attached to *two* nodes in the network and the failure event causes one of these adjacencies to fail. In the clique graphs, node 0 and n were the attachment points. In addition, we force the path through n to be the least preferred path in the network. In the other models, we repeat the simulations with different pairs of attachment points.

The convergence process for the *fail-over* case can be understood as follows: when a link incident to the dummy node fails, the node on the other side of the link sends withdrawals to its neighbors, which are then propagated through the network. This causes some of the nodes in the graph to switch to an alternate path. Note that some nodes might already be using the alternate path, and will not be affected by the failure. In the clique topologies, *all* nodes are forced to choose an alternate path, since the fail-over backup path has the lowest preference.

In EPIC, when the withdrawal is generated, the nodes that receive it remove *all* invalid paths immediately, and the convergence time is determined by the time taken to distribute the alternate path to nodes that don't already have it. Fig. 5(c) and 6(c) show the relative performance of the protocols in the clique topologies, while Figs. 7(a) and 7(b) illustrates the same for the Waxman topologies.

It is clear from these graphs that EPIC performs better than the other protocols. This is to be expected, since in the cases of Ghost Flushing and BGP, once the preferred path is withdrawn, the routers begin to "explore" the longer paths that contain the failed edge. Another interesting point to note with this set of graphs is that the convergence time of Ghost Flushing is quite close to that of BGP, and in some cases *worse*! (while it was always better in the *fail-over* case). This seems to suggest that when there are alternate paths, it might be counter-productive to be aggressive in withdrawing paths. The additional withdrawals may actually delay nodes from learning the *valid* alternate path.



(a) Convergence Time

(b) Messages generated

Figure 7: Performance in Waxman graphs, *fail-over* scenario. Bottom curve corresponds to EPIC, topmost curve corresponds to BGP.

9 Related Work

The notion of "tagging" withdrawals with location information was first mentioned in [MVW+00]. More recent work discussed in [PAN+03, ZAL03] build on the same idea. While all these ideas share some similarity with ours, i.e., the notion of attaching "event information" to BGP withdrawals, they not take into account the complexity introduced by BGP. In particular, *all* of the above ideas assume a network model where each AS has a single router and adjacent ASes share a single edge. However, as discussed in Sec. 2.3, an AS *cannot* be modelled as a single node, since the routers in the AS are independent entities making independent routing choices with different information available to each. To the best of our knowledge, ours is the first solution to use a realistic model of BGP and Internet topology.

In [PZW+02], the authors discuss a solution based on "consistency rules" in announcements from a set of neighbours. The drawback is that these rules are applicable only in specific settings, for example, when the paths from from different neighbors have a mutual dependency. *Ghost Flushing*, described in [BBAS03], is a simple idea to reduce convergence following a failure. The underlying idea is to aggressively send withdrawals, forcing the invalid routes to be flushed from the network. While this idea is conceptually very simple, it does not really prevent *path exploration*, but instead tries to speed up the process.

10 Conclusions

In this paper, we describe why path exploration, which is the root cause of slow convergence in BGP, cannot be addressed effectively within the existing BGP framework. We then proposed a simple, novel mechanism—*forward edge sequence number*—to annotate the AS paths with path dependency information. Then we described *EPIC*, an *enhanced* path vector protocol, which prevents path exploration after failure, and discussed some of the additional overhead. To the best of our knowledge, *EPIC* is the first solution to be shown to work in an extremely general model of Internet topology and BGP operation. Using theoretical analysis and simulations, we demonstrated that *EPIC* achieves a dramatic reduction in routing convergence time, as compared to BGP and other existing solutions.

References

[BBAS03] Anat Bremler-Barr, Yehuda Afek, and Shemer Schwarz. Improved BGP Convergence via Ghost Flushing. In *Proc. IEEE INFOCOM*. IEEE, Mar 2003.

[bgp] BGP path selection algorithm. `http://www.cisco.com/warp/public/459/25.shtml`.

[bri] BRITE: Boston University Representative Internet Topology. `http://www.cs.bu.edu/brite/`.

[CDZK04] Jaideep Chandrashekar, Zhenhai Duan, Zhi-Li Zhang, and Jeffrey Krasky. Limiting Path Exploration in Path Vector Protocols. Technical report, University of Minnesota, 2004. `http://www.cs.umn.edu/~jaideepc/papers/epic-tr.pdf`.

[CGH02] Di-Fa Chang, Ramesh Govindan, and John Heidemann. An empirical study of router response to large BGP routing table load. In *ACM Sigcomm Internet Measurement Workshop*, November 2002.

[coo02] The COOK Report on Internet. `http://www.cookreport.com`, November 2002.

[Gri02] Tim Griffin. What is the sound of one route flapping. Network Modeling and Simulation Summer Workshop, Dartmouth, 2002.

[GW99] Timothy G. Griffin and Gordon Wilfong. An Analysis of BGP Convergence Properties. In *Computer Communication Review*, volume 29. ACM, October 1999.

[LABJ01] Craig Labovitz, Abha Ahuja, Abhijit Bose, and Farnam Jahanian. Delayed Internet Routing Convergence. *IEEE/ACM Trans. Netw.*, 9(3):293–306, 2001.

[LAWS01] Craig Labovitz, Abha Ahuja, Roger Wattenhofer, and Venkatachary Srinivasan. The Impact of Internet Policy and Topology on Delayed Routing Convergence. In *Proc. IEEE INFOCOM*, pages 537–546, 2001.

[LMJ99] Craig Labovitz, G. Robert Malan, and Farnam Jahanian. Origins of Internet Routing Instability. In *Proc. IEEE INFOCOM*. IEEE, Mar 1999.

[MGVK02] Zhuoqing Morley Mao, Ramesh Govindan, George Varghese, and Randy Katz. Route Flap Damping Exacerbates Internet Routing Convergence. In *SIGCOMM*, August 2002.

[MVW+00] Madan Musuvathi, Srinivasan Venkatachary, Roger Wattenhofer, Craig Labovitz, and Abha Ahuja. BGP-CT: A First Step Towards Fast Internet Fail-Over. Technical report, Microsoft Research, 2000.

[PAN+03] Dan Pei, Matt Azuma, Nam Nguyen, Jiwei Chen, Dan Massey, and Lixia Zhang. BGP-RCN: Improving BGP Convergence through Root Cause Notification. Technical Report 030047, UCLA, October 2003.

[PZW+02] Dan Pei, Xiaoliang Zhao, Lan Wang, Daniel Massey, Allison Mankin, S. Felix Wu, and Lixia Zhang. Improving BGP Convergence Through Consistency Assertions. In *Proc. IEEE INFOCOM*. IEEE, 2002.

[RLH03] Yakov Rekhter, Tony Li, and Susan Hares. A Border Gateway Protocol 4 (BGP-4). `http://www.ietf.org/internet-drafts/draft-ietf-idr-bgp4-23.txt`, December 2003. Internet Draft.

[VGE99] Kannan Varadhan, Ramesh Govindan, and Deborah Estrin. Persistent route oscillations in inter-domain routing. *Computer Networks*, 32(1):1–16, 1999.

[Vie00] Route Views. University of Oregon Route Views Project. `http://antc.uoregon.edu/route-views/`, 2000.

[ZAL03] Hongwei Zhang, Anish Arora, and Zhijun Liu. G-BGP: Stable and Fast Convergence of the Border Gateway Protocol. Technical Report OSU-CISRC-6/03-TR36, Ohio State University, June 2003.