Limiting Path Exploration in BGP

Jaideep Chandrashekar
Dept of Computer Sci. and Engg.
University of Minnesota,
Minneapolis, MN
Email: jaideepc@cs.umn.edu

Zhenhai Duan
Dept of Computer Science
Florida State University
Tallahassee, FL
Email: duan@cs.fsu.edu

Zhi-Li Zhang
Dept of Computer Sci. and Engg.
Univ. of Minnesota,
Minneapolis, MN
Email: zhzhang@cs.umn.edu

Jeff Krasky
Dept of Computer Sci. and Engg.
Univ. of Minnesota,
Minneapolis, MN
Email: jkrasky@cs.umn.edu

***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. We then develop an enhanced path vector algorithm, *EPIC*, 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.**

I. Introduction

The Internet is a collection of independently administered *Autonomous Systems* (ASes) glued together by the Border Gateway Protocol (BGP) [1], 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 [2]. The root cause of this slow convergence is the *dependency* among paths announced through the network, leading to *path exploration*: when a previously announced path is withdrawn, other paths that *depend* on the withdrawn path (now *invalid*) may still be chosen and announced, only to be removed later one by one.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, in the worst case, a path vector routing protocol can explore as many as $O(n!)$ alternative routes before converging. Addressing path exploration within the framework of BGP is particularly challenging: AS paths carried in 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 affecting the performance of applications such as VoIP, streaming video, online gaming, etc. Second, the additional protocol activity increases 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 [3]. Third, normal path exploration may be incorrectly identified as instability (i.e., flapping routes), triggering damping mechanisms at routers [4]. Lastly, it complicates the task of identifying the root-causes of routing updates, essential in understanding inter-domain routing dynamics [5].

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. Using this mechanism, we develop an enhanced path vector routing protocol, *EPIC*, which limits path exploration and thereby 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; 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. II briefly reviews BGP operation and illustrates the path exploration problem. In Sec. III 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. IV,

This work was supported in part by the National Science Foundation under the grants ANI-0073819, ITR-0085824, CNS-0435444 and a Cisco URP gift grant. Any opinions, findings, and conclusions or recommendations expressed in this paper are those of the authors and do not necessarily reflect the views of the National Science Foundation.

along with correctness results. Sec. V lists some analytical results for *EPIC* and simulation results are presented in Sec. VI. Finally, we review some related work in Sec. VIII and conclude in Sec. IX.

II. 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 (not just BGP) and describe why it is particularly hard to address, in the context of BGP.

A. 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 adjacent ASes may have more than one eBGP session. We now briefly describe the relevant operation at a BGP router (see [1] 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 (in AS 0).

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 all the ASes 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 locally defined policies — to select a single "best" route from this set [6]. 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, but not to an internal neighbor in the same AS, the AS path carried in the announcement is prepended with the ASN of the local AS.

B. Path Exploration

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 in path vector protocols, the *path vectors* are used to 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.

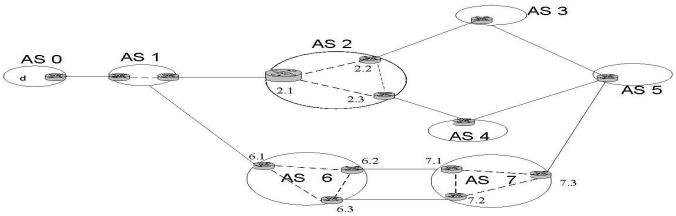


Fig. 1. BGP and Path Exploration. Solid lines represent eBGP sessions, while dashed lines indicate iBGP sessions.

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] (preferred in that order)..

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 route" 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. Previously, Labovitz *et.al.* showed that, in the worst case, as many as $O(n!)$ alternate paths may be explored after a failure [7]. In practice however, in todays Internet, such a worst-case scenario is rare: common routing policies reduce the number of available routes, and protocol timers limit how fast updates can be sent — both of which have a beneficial effect. Nonetheless, path exploration can still adversely impact performance. 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, with very high link speeds and rich connectivity.

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.

First, we introduce some notions that will be useful in the rest of the paper. We call a network event (e.g., link failure or repair, router crash, BGP session reset, etc.) that affects an eBGP session as an *external* event, and that which affects an iBGP session between two internal routers an *internal* event. We also distinguish a *network event* from a *routing* event, which refers to the generation of a route update by a BGP

router. Note that there is *not* a one-to-one correspondence between network and routing events. For instance, a network event such as a physical link failure or router crash may affect many BGP sessions, trigerring multiple routing events. Using this notion of routing events, we could also distinguish between the (routing) *event originator* and *event propagator*, and correspondingly, *primary* (routing) events and *secondary* (routing) events. The (routing) event originator is the router which, upon detecting an (external) event — BGP session failures (or recovery), etc., — generates a new route update. Thus, the generation of this new route update is the primary (routing) event. A router receiving a route update may further propagate the "event" by generating a subsequent route update, i.e., a secondary event; thus we will refer to it as an *event propagator*. For clarity, we will refer to the (eBGP or iBGP) sessions between routers as (external or internal) *edges*. Thus, $\langle u, v \rangle$ represents a BGP session between *routers* u and v. When the context is clear, we also use the same notation to denote an edge between adjacent ASes (for example, if there is a single router in each of AS 1 and AS 2, there is no confusion if we write $\langle 1, 2 \rangle$). Neighboring ASes may have multiple eBGP sessions between them; in such cases we call these *minor edges*.

C. Path Exploration and BGP Complexity

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.[1] The devil being in the details, this summarization conceals information that would have made it possible to detect path dependencies. In the following, revisiting the topology in Fig. 1, we illustrate how different failure events can generate the *same* updates, complicating the task of detecting path dependency in BGP.

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. Due to this failure, which is detected by 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. Note that this primary event should affect both routes that AS 5 learned earlier from 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 in the routing table (at AS 5) is $[3210]$.

In both these distinct scenarios, AS 3 will send a withdrawal to AS 5, (implicitly) invalidating the same route, i.e., $[210]$. 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.

Now consider a third, more complicated scenario: 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 with that where the edge between AS 0 and AS 1 fails, which also causes 7.1 to send the withdrawal for $[610]$ 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 events may occur close in time. Due to the general complexity of AS topology and the varied propagation delays along different paths, updates for events may arrive at routers in a different order from which the events occurred. As an example, 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 $[10]$. Now, suppose the delays in the network are such that the withdrawal and subsequent re-announcement arrive at AS 5 through AS 3 faster than the (first) withdrawal travelling 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.

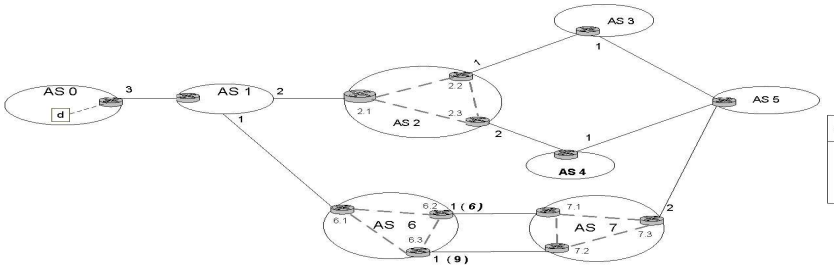
These examples clearly illustrate that the AS paths, carried with BGP routes, do not contain sufficient information to correctly distinguish valid and invalid paths, which is a critical requirement to suppress the exploration of obsoleted paths. Clearly, to address this effectively, we need to incorporate *additional* information into route updates that will correctly capture the dependencies between invalidated paths, and also allow route updates corresponding to newer events to be distinguished from older ones. In addition, such a mechanism should not require an AS to expose detailed (or internal) connectivity information, nor impose undue processing, memory or communication overheads on a router. In the next section we introduce the notion of *forward edge sequence numbers*, which satisfies these requirements. Using this AS path "annotation", routers can identify *all* routes that are rendered obsolete by some failure event, and invalidate them *all at once*, significantly improving the protocol convergence time.

III. 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 *forward edge sequence numbers* (or *fesns*) to capture the "state" of a forward edge. There are two different types of

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



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) }

Fig. 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.

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 same neighbor AS).

Formally, at any AS, say X, corresponding to each neighbor, 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, which has the integer sequence number n. Note that 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 responsible for incrementing the sequence number.[2]

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.[3] The same operation is performed at every router along the way and consequently, a route contains an ordered list of *major fesns*, called the *fesnList* of the route. Importantly, 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.

When there are multiple minor edges between neighboring ASes, they are *all* 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 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. $(x' : y', k')$ and $(x'' : y'', k'')$. However, they are both associated with the *same* AS level *major fesn*. Importantly, *minor* and *major* fesns are incremented the same way—when the corresponding edge is restored. A key difference is that *minor fesns* are only carried in internal routing updates and between the corresponding neighbors, 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, but stripped out from any updates sent to a different AS, such as when router 7.3 propagates the 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) *fesns* for each *forward edge* are indicated along the edge (the numbers in parenthesis are *minor fesn* values). For simplicity, we abstract consider only the relevant parts of the update message exchanged between routers and denote it [ASPATH]{fesnList}; this captures 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 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 that the AS path is identical in the updates, but the *fesnList's* are distinct.

When routes are advertised internally, the *fesnlist* is carried unchanged. Hence both 2.2 and 2.3 will receive identical route announcements, i.e. [10]{(1:2,2)(0:1,3)}. When these routers, in turn, propagate the route to their own 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 and propagated through the network, the routing table at AS 5 is as shown in the table in Fig. 2.

Following a failure, when an *event originator* generates (initiates) a route withdrawal, it will insert the *fesnList* of the invalid route into the withdrawal. When a neighbor (an *event propagator*) receives this and generates a subsequent routing update, it attaches the (original) withdrawal *without change.* In other words, an *event propagator* will forward an *exact copy* of the withdrawn *fesnList* it receives. If the *propagator* selects a new *best route* after processing the withdrawal,[4] the *original* withdrawal is "piggybacked" onto the (resulting) route announcement. Thus, every router that receives an update after the failure will see the original *fesnList* inserted by the *originator*. However, the invalid (or withdrawn) route may not directly correspond to an AS path announced by a router. To make this distinction clear, we shall call the actual path described in the *fesnList* as a "path-stem"since, intuitively, all the invalid paths are essentially "branches" from this stem.

[2]However, in a few special cases, to force consistency, we also require AS Y to *independently* increment the *fesn*.

[3]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.

[4]This is the case when other alternate, valid paths exist.

In the rest of this section, we revisit the failure scenarios described in section II-B and illustrate how the *fesnlist* attribute can be used to identify the obsolete routes following a failure, avoiding the previously described difficulties (in Sec. II-B).

External edge $\langle 1, 2\rangle$ ***fails:*** This is detected by router 2.1 which *originates* a route withdrawal, sent to its internal peers 2.2 and 2.3. Following previous notation, we describe a withdrawal message as `W:[AS PATH]` {`fesnList`}. Then the withdrawal sent by 2.1 is W:[10]{(1:2,2) (0:1,3)}.[5] 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 (in Fig. 2) match, and will be removed. However, note that the third route, corresponding to AS path [7610], does not match and will be retained in the routing table. Thus, the *first* withdrawal received at AS 5 will at once invalidate routes learned through AS 3 and AS 4, both of which depend on the failed edge. This is in contrast to BGP, where 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 the router in AS 1 will increment the *fesn* for the edge. The repair also triggers a primary routing event, i.e., the route announcement [10]{(1:2,3)(0:1,3)}, sent to router 2.1. At 2.1, this *new* route[6] is installed in the routing table. The route is then exported to 2.2 and 2.3, which in turn forward the route to AS 3 and AS 4. 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 this 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; the utility of the *fesn* concept becomes clear in this example. 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 easily differentiate the same AS path announced by 2.2 to different neighbors. Thus, the router correctly remove the affected (now invalid) and still retains the valid route.

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.

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, [610]. Thus, when 7.3 receives the withdrawal from 7.1, the route learned from 7.2 should not be invalidated. While both the AS path and *fesnLists* are common, note that that *minor fesn* attribute is different for routes learned over the two edges.

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 the end. At 7.3, both *fesnList* and *minor fesn*, present in the withdrawal, are used to identify invalid routes. In the present example, 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 learned from 7.2, which does not match the withdrawn *minor fesn*.

Subsequently, 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 is required to generate a subsequent route announcement to indicate this to its downstream neighbors.

The above 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.

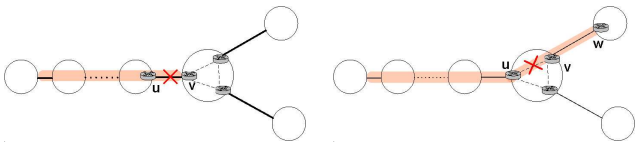
IV. EPIC – Detailed Description

In this section, we describe EPIC in detail. EPIC is an enhanced path vector protocol which supports the new *fesnList* route attribute that we defined previously. A central notion in our solution is the distinction between how *event originators* and *propagators* operate. In this section, we describe the operation of each entity. Subsequently, we establish some correctness properties of our path vector protocol and discuss the additional overhead required at routers to support it.

To begin with, we establish some notation that is used in the following discussion. 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 $[u \to v]$, we mean the actual route announced by u to v. Following this notation, we reference the ASPATH and *fesnList* attributes associated with each route as $\mathsf{u}^*.\mathsf{aspath}$, $[u \to v].\mathsf{aspath}$, $\mathsf{u}^*.\mathsf{fesnList}$ and $[u \to v].\mathsf{fesnList}$. In the following, we discuss the detailed operation at *event originators* and *event propagators* in response to failure and repair events. In order to present the main concepts clearly, we make two simplyfying assumptions: 1) all internal routers within an AS are fully meshed, and 2) there is at most a single edge (peering

[5] As a technical detail, the withdrawal will not explicitly contain the AS Path, which is *embedded* in the *fesnList* itself. We include it in the description to make the examples easier to follow.

[6] Notice the incremented value for the *fesn* corresponding to the edge $\langle 1, 2\rangle$.



(a) External Failure (b) Internal Failure

Fig. 3. Intuition for selecting *fesnList* after failure. Path-stems that are rendered invalid are different for internal and external failures. In each figure, the shaded segment describes the "path stem" after the corresponding edge has failed.

session) between two adjacent ASes. However, in reality, one of both of these may not hold. In a longer version of this paper, we remove these restrictions and present a complete solution [8].

Event Originator: A router becomes an *event originator* if a failure or repair is detected on an adjacent edge. First, we discuss the operation after failure and subsequently the repair scenario. Importantly, in either case, a router will generate a route update (initiate a routing event) *only if the event causes a change in the best route.*[7]

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. If no other alternate route exists, this will be a route withdrawal, otherwise a route announcement. Importantly, in the latter case, a withdrawal is attached to the announcement that is sent by the router. In either case, the withdrawal contains the *fesnList* of the invalidated route. The detailed algorithm after a failure is presented[8] 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. We consider each case separately in the following.

First, consider the case where the failed edge, i.e., $\langle u, v\rangle$, is external. The intuition behind selecting the *fesnList* is shown in Fig. 3(a). As shown in the figure, 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 described in the *fesnList* associated with the route announced by u to v, i.e., $[u \rightarrow v]$.fesnList. Also, 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 somewhat different if $\langle u, v\rangle$, the failed edge, is internal. Note that the "edge" $\langle u, v\rangle$ is *not embedded in any AS path or fesnList*. However, the concept of "forward edges" can address this, and the mechanism can be explained using Fig. 3(b): here, when the internal edge $\langle u, v\rangle$ fails, the affected path-stem, shown by the shaded area, includes the *forward edge* $\langle v, w\rangle$. Recall that the *fesnList* in the route sent from v to w includes the *fesn* for this edge. Thus, $[v \rightarrow w]$.fesnList describes an affected path-stem. When v has multiple external neighbors, 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. In other words, different (withdrawal) routing events are initiated for the affected path-stems. To correctly handle the situation, v originates a (failure) routing event for each external neighbor: if w is an external neighbor, then v will generates a withdrawal containing $[v \rightarrow w]$.fesnList and send it to w. Note that this is possibly attached to a route announcement if w selects an alternate route following the failure.

[7]So for example, if the best route used by the router was *not* learned over the failed edge, or if a new route learned over the repaired edge does not become the best route, then no new event is originated.

[8]We abstract the actual decision process at a router into the `SELECT_BEST_PATH()` procedure.

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

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

Repair: Recall that *repair* events also trigger route updates and the corresponding operation is described in Alg. 2. In contrast to the failure scenario, when $\langle u, v\rangle$ is repaired, the identity of the *event 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 *event originator* is the that which "exports" the event out of the AS.

If $\langle u, v\rangle$ is external, then u is the *event originator*. Upon detecting 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 any route previously learned from v.

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. When a route update is received at

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

```
if ⟨u, v⟩ is external then {u is the originator}
   @ router u:
   increment fesn(⟨u, v⟩)
   ANN = v*
   send (ANN) to v
else {internal edge repair; v is the originator}
   @ router v:
   include [u→v] in candidate set
   v* = SELECT_BEST_ROUTE( )
   if v* has changed then
      ANN = v*
      for all w ∈ NEIGHBOR(v) do
         increment fesn(⟨v, w⟩)
         ANN = v*
         send (ANN) to w
      end for
   end if
end if
```

an *event propagator*, it is processed as follows: if the update contains a withdrawal, then all routes in the routing table which *depend* on the *fesnList*—carried 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. If the best route has changed after this operation, then a (secondary) route announcement is generated by the *propagator*. 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, the original withdrawal is attached. The important point here is that a withdrawal received by the *propagator*, if it causes a route change, is forwarded without modification. Thus, every router that learns of the failure event uses the exact same *fesnList*, generated by the *originator*, to invalidate routes.

The crucial step in processing a withdrawal is to correctly identify the subset of routes that *depend* on the withdrawn path-stem. Here, the structure of the *fesnList* attribute makes it easy to identify *dependent* routes.

Let $f_1 = \langle (a_1 : b_1, n_1), \ldots (a_k : b_k, n_k) \rangle$ be the *fesnList* contained in the withdrawal, and $f_2 = \langle (a'_1 : b'_1, n'_1), \ldots (a'_l : b'_l, n'_l) \rangle$ be the *fesnList* of any route in the candidate set, with $k \leq l$. Then, we can state that the f_2 *depends on* f_1, the withdrawn path-stem, 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 because the *fesn* value for an edge is monotonic; hence larger *fesn* values indicate "newer" information. Thus, any routes in the candidate set that satisfy the predicate are invalid and will be *excluded* from the subsequent decision process.

Before we conclude this section, we briefly touch upon some of the issues that arise when either of the assumptions made at the beginning fails, i.e., when the internal routers in an AS are organized into a hierarchy using Route reflection [9], or when there are multiple peering sessions between adjacent ASes. When either of these assumptions fails, there is a "loss of visibility" within the AS. For instance, suppose there are multiple peering sessions between two ASes and one of them is repaired (following a failure). To manage the *major fesn* correctly, the router (adjacent to the affected edge) needs to know the status of *all* other sessions between the same neighbors.[9] Additional mechanisms are required to ensure that the *major fesns* are consistent even with this loss of visibility. For example, the concept of *minor fesns* is used to handle adjacent ASes with multiple peering sessions. Due to a lack of space, we only discuss the more straightforward setting in this paper. A complete specification can be found in a longer version [**?**].

A. Correctness

In this section, we establish some theoretical properties of our enhanced path vector protocol (EPIC). In particular, we show that following a single failure (routing) event, no router in the network will select (and subsequently propagate) an invalid route, i.e., a route that depends on a failed edge.

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* edge $\langle u, v \rangle$ fails, the withdrawal contains $[u \rightarrow v]$.fesnList. This explicitly embeds information for the $\langle u, v \rangle$, 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. Also, 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 invalidated path are excluded from the decision process (to select an alternate route). Also, no valid route is invalidated.

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

Now suppose that a *valid* route, say r_v, associated with an *fesnList*, $r_v.fesnList = \langle (a'_1 : b'_1, n'_1), \ldots, (a'_k : b'_k, n'_k) \rangle$ is invalidated when a withdrawal, r_w, associated with $r_w.fesnList = \langle (a_1 : b_1, n_1), \ldots, (a_k : b_k, n_k) \rangle$ is received at the event propagator. Clearly, $k \leq l$ and $\langle a_i, b_i \rangle = \langle a'_i, b'_i \rangle$, for $i = 1, \ldots, k$ as otherwise, r_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 r_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 r_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 r_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.

[9]Recall that the *major fesn* will be incremented only if all the *minor* edges had failed and the particular link was restored subsequently.

	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

TABLE I

ADDITIONAL MEMORY REQUIREMENTS TO STORE THE *fesnlist* ATTRIBUTE.

Proof: Suppose that the edge $\langle u, v\rangle$ fails and a single (failure) routing event is originated. Without loss of generality, we can assume that the failure affects the best route at v (the event originator), forcing 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. ■

B. Overhead

In this section, we briefly discuss the additional overhead introduced by *EPIC*, in memory and communication. Recall that the main component of our scheme is the additional *fesnList* attribute that is carried in route updates and withdrawals.

In the following, we estimate the additional memory required at a router to store the new route attribute. 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.[10] Also, a router receives 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.[11] Furthermore, route aggregation tends to reduce number of prefixes, so we expect the actual overhead to be lower.

[10]Note that this is not the same as the "diameter". In BGP, locally defined policy can force a router to chose a longer paths.

[11]In a recent snapshot of the routing table taken from Route-Views, we computed $L = 12$, while the median AS Path length was 4.

In our description so far, the *fesns* explicitly listed the associated edge. However, correct operation only requires that *fesns* be distinct, i.e., that *fesn* for an edge $\langle X, Y\rangle$ is unique. Correspondingly, it is possible to use a "compressed" *fesn* representation. 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. In fact, there may be other, more effective compression schemes that can reduce this further. However, a detailed discussion of such schemes is outside the scope of this paper.

Table I lists the estimated memory requirements to store the *fesnList* attribute. Note that pre-pended ASes have been removed (since they don't affect the length of the *fesnList*). Also, the estimates assume that each *fesn* is encoded with 4 bytes. The data for the "core router" was obtained from Route-Views [10], and data for the campus router is taken from the border router at the University of Minnesota which has four upstream providers.

In addition to storing the *fesnList* attribute in the routing table, a router needs to keep track of *major* (and *minor*) *fesns* associated with adjacent "forward edges" or BGP sessions. Since only the *fesn* values are stored, an additional $O(4dP)$ bytes of memory is required.

With respect to the communication overhead, first note that EPIC will not generate any more routing updates than BGP. Also, every *EPIC* routing announcement and withdrawal carry the additional *fesnList* attribute, which slightly increases the size of update messages. 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.

V. PERFORMANCE ANALYSIS

In this section, we derive upper bounds for the time and communication complexity of EPIC and contrast it with the performance of BGP and Ghost Flushing [11]. The latter is a path vector protocol variant that attempts to speed convergence by aggressively distributing withdrawals in the network after a failure.

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 IV

NOTATION USED IN THE ANALYSIS.

In order to make the analysis tractable, we use the discrete-time synchronized model described in [7]: in each time step, a node processes *all* of the messages received in the last stage and selects a single "best path". Then, if the best path has changed, it is exported to all adjacent neighbors. Withdrawal messages are handled differently in each scheme: in BGP, withdrawals are sent only if a node has no other valid paths to the destination; in Ghost Flushing (GF), a node generates a withdrawal message whenever it receives

	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 II
PERFORMANCE BOUNDS FOR FAIL-DOWN.

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

TABLE III
PERFORMANCE BOUNDS FOR FAIL-OVER.

a route announcement for a longer path, even if alternate routes exist in the routing table.[12] Finally, in EPIC, a node forwards a withdrawal message *only* if the received withdrawal invalidated the best path at the node.

We analyze the bounds in two different failure scenarios—*fail-down and fail-over*. In a *fail-down* event, the network becomes partitioned after the event. In other words, following 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 following analysis is enumerated in Table IV.

A. Fail-down case

The upper bounds in the fail-down scenario are summarized in Table IV-B. The performance bounds for BGP and Ghost Flushing follow directly from the results in [11], [12], 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 the 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 following the failure (in the period of convergence). 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 $(|E|-1)$. ■

[12]The underlying intuition is that announcements for longer paths may indicate path exploration.

B. Fail-over case

Now, 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, we fix a particular node as the destination. When a link fails, and alternate paths (to the destination) exist, we can partition the set of nodes into $\hat{V}$ and $V-\hat{V}$ in the following manner: if the best path at a node becomes invalid after the failure, it is in $\hat{V}$. Also, all of this node's neighbors are in $\hat{V}$. All other nodes are in $V-\hat{V}$.

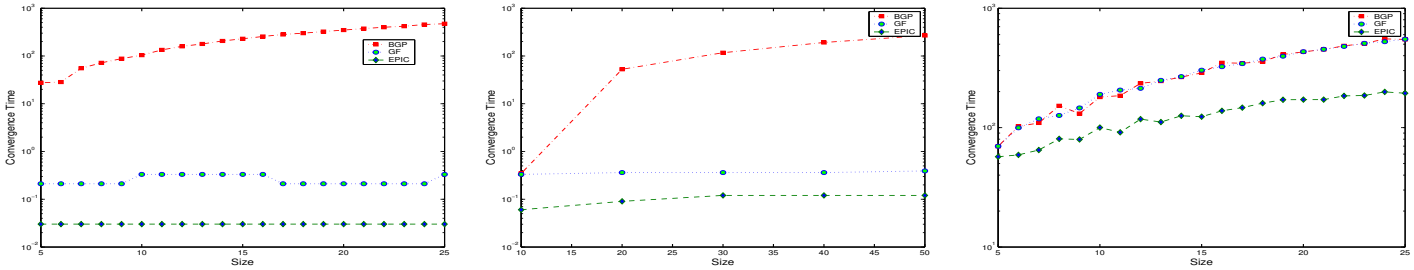
Informally, $\hat{V}$ contains nodes whose *preferred path* is affected by the failure, along with their adjacent neighbors. Clearly, the nodes in $V-\hat{V}$ are not affected by the failure and will not take part in the convergence process i.e., they will neither receive nor propagate any updates, since their best path does not change after the failure. Then, corresponding to $\hat{V}$, we can imagine a "zone of convergence" in G, defined as the induced graph $G[\hat{V}]=(\hat{V},\hat{E})$. Finally, by L, we take to mean 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 $G[\hat{V}]$. Also, to simplify the analysis, we ignore the effects of BGP policy and assume that shorter paths are preferred.

The performance bounds for each of the the protocols in the *fail-down* case are summarized in Table IV-B. Due to a lack of space, we only present the results for EPIC in the following and relegate the remaining to the longer version of this paper [8].

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.

Proof: [sketch] From the description of the protocol in Sec. IV, note that a router will does not select and propagate a path unless its 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 re-convergence process (where the alternate paths are being propagated), a node might wait up to Δ time before



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

Fig. 4. Convergence Time in Clique and Waxman topologies. In all the graphs, the bottom curve represents the performance for EPIC.

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.

In order 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 $\Delta(h+\Delta)\frac{1}{\Delta}$ messages are sent across *each edge*, and the message complexity is bounded by $(\hat{D}(h+\Delta)/\Delta)(|\hat{E}|-1)$. ■

VI. 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 with that of BGP and GhostFlushing [11]. We used two different topology families for our simulations—Cliques and Waxman Random Graphs ($\alpha = 0.3, \beta = 0.4$) The latter two topology families were generated using the Brite topology generator [13]. In each generated topology, links are assigned a uniform propagation delay of 300ms. Furthermore, for each protocol, we used an `MinRouteAdvertiseInterval` value of 30 seconds, which is the BGP default for a popular router vendor.[13]

Using each topology, we simulated both *fail-down* and *fail-over* scenarios. In this section, we discuss two performance metrics: convergence time and message complxexity. In the following, we plot these metrics for both types of scenarios.

A. Results for fail-down

In this scenario, for 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. Simulations were repeated multiple times with different random seeds, and the average performance is plotted in the graphs.

In Figs. 4(a) and 4(b), we plot the network size against the convergence time, with the y-axis shown in logscale, for the clique and waxman topologies (respectively). In both figures, the bottom curve represents the performance of EPIC, the top curve corresponds to BGP and the middle curve is the performance of GhostFlushing. Clearly, EPIC performs far better than either of the other protocols. Notice that in the clique topology, i.e., Fig. 4(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, in BGP and Ghost Flushing, the alternate (invalid) paths are 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. 5(a) and 5(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. 5(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).

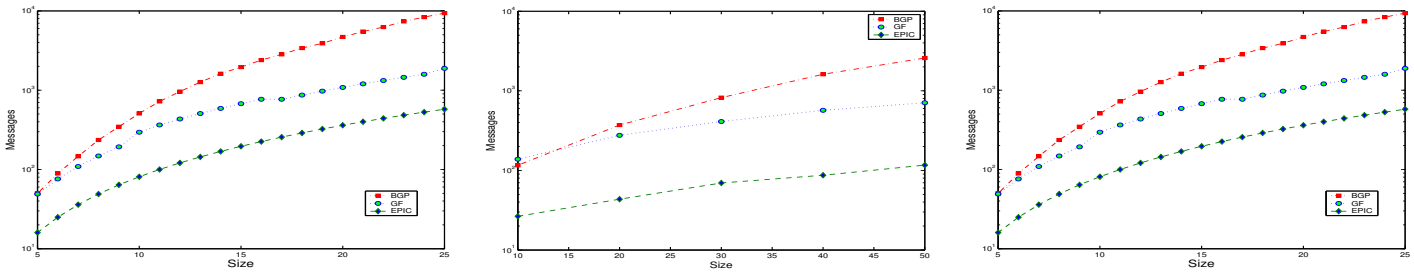
Unlike the clique topologies, it is difficult to analytically estimate the convergence time or the message volume in the Waxman topologies, since the different sized graphs are independently generated. Nevertheless, the graphs plotted in Figs. 4(b) and 5(b) clearly illustrate that EPIC generally performs 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.

B. 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 waxman topologies, 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

[13]This corresponds to Δ, defined in the previous section.



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

Fig. 5. Number of Messages generated during convergence. Note that 5(a) and 5(b) correspond to *fail-down*. In all three figures, the bottom curve corresponds to the performance in EPIC.

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. 4(c) and 5(c) show the relative performance of the protocols in the clique topologies, while Figs. 6(a) and 6(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.

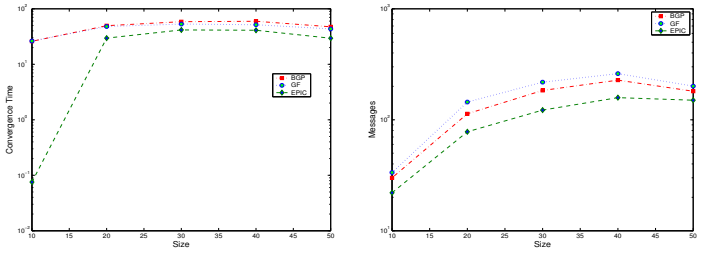
VII. Implementation and Deployment

In this section, we briefly discuss how to implement the *fesnList* as an actual BGP route attribute (to be carried with route advertisements) and how we can deploy EPIC incrementally into the existing BGP system.

The *fesnList* can can be implemented as a new *optional transitive* attribute. Specifically, we define the `fesnList` route attribute is defined as a list of 4 byte *fesn* attributes. The higher order 2 bytes of the *fesn* encode $X(xor)Y$ (for AS-AS edge $\langle X, Y \rangle$), where X and Y are 2 byte AS Numbers. The lower order bytes encode the sequence number.[14]

Note that EPIC preserves the existing BGP protocol semantics with a single exception—withdrawals carry an `fesnList` attributeIn order to ensure compatibility with legacy routers, when an EPIC router is sending a route withdrawal to a non-EPIC router, it is required to first *strip* out the `fesnList` from the withdrawal. Clearly, route announcements do not cause a problem—the `fesnList` route attribute is simply

[14]Another possibility would be to reuse the extended communities attribute that is already defined [14].



(a) Convergence Time (b) Messages generated

Fig. 6. Performance in Waxman graphs, *fail-over* scenario. Bottom curve corresponds to EPIC, topmost curve corresponds to BGP.

processed as any other *optional, transitive* attribute. It should stressed that that our solution does not "break" existing BGP; the `fesnList` is used *in addition* to the other well-defined attributes in BGP. When a withdrawal message is sent to a BGP router (with the `fesnList` removed), it is a well formed BGP message and will be processed as a normal BGP withdrawal. However, when an EPIC route announcement is sent to a non-EPIC router, the `fesnlist` is no longer contiguous and hence cannot be used anymore to invalidate routes based on withdrawals. Thus, correct BGP operation is preserved, though there might not be any benefit in terms of network convergence.

Obviously, the faster network convergence that is enabled by EPIC will be apparent only when it is deployed system wide (or at least in some large contiguous region), there are other independent benefits when deployed in smaller, localized settings. For example, note that the rate of change of an *fesn* measures the instability of the associated edge—the sequence number is incremented when the edge flaps—pointing to an efficient and correct way to identify flapping routes, which overcomes the drawbacks in previous*indirect* schemes [4].

A more interesting application of EPIC is in enabling operators to perform "root cause analysis" of BGP updates [5]. Within an AS, annotating routes taht are being distributed can be very useful in tracking the location of routing changes. For example, if we think of *minor fesns* simply as "edge labels", it becomes easy to differentiate between updates caused by instability on the peering links or that which is propagated from outside the AS. Note that without such an "annotating" mechanism, performing the same analysis can be quite complicated [15]. Moving beyind a single AS, if we consider a larger deployment, the *fesnList* can be used to determine the location of routing events that trigger routing

changes. Thus, a key side effect of deploying EPIC on the Internet is to provide a much needed diagnostic capability.

Given the obvious *local* benefits, we believe that operators would be motivated to deploy EPIC within their own networks, filtering out the EPIC attributes at the edge. Subsequently, when two or more neighboring ASes run EPIC on their networks, the scope of operation can be easily enlarged by turning off the filters and allowing the new attributes to go across. Eventually, as the deployment base grows, we will see the more apparent global benefit, i.e., reduced convergence time.

VIII. Related Work

The notion of "tagging" withdrawals with location information was first mentioned in [16]. More recent work discussed in [17], [18] 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 do not address 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. II-B, 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 [19], 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 [11], 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.

IX. 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.

Acknowledgement

We are grateful to Jennifer Rexford for her very valuable suggestions. Also, we would like to thank the anonymous reviewers for their helpful comments.

References

[1] Y. Rekhter, T. Li, and S. Hares, "A Border Gateway Protocol 4 (BGP-4)," http://www.ietf.org/internet-drafts/draft-ietf-idr-bgp4-23.txt, December 2003, internet Draft.

[2] C. Labovitz, G. R. Malan, and F. Jahanian, "Origins of Internet Routing Instability," in *Proc. IEEE INFOCOM*. IEEE, Mar 1999.

[3] D.-F. Chang, R. Govindan, and J. Heidemann, "An empirical study of router response to large BGP routing table load," in *ACM Sigcomm Internet Measurement Workshop*, November 2002.

[4] Z. M. Mao, R. Govindan, G. Varghese, and R. Katz, "Route Flap Damping Exacerbates Internet Routing Convergence," in *SIGCOMM*, August 2002.

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

[6] "BGP path selection algorithm," http://www.cisco.com/warp/public/459/25.shtml.

[7] C. Labovitz, A. Ahuja, A. Bose, and F. Jahanian, "Delayed Internet Routing Convergence," *IEEE/ACM Trans. Netw.*, vol. 9, no. 3, pp. 293–306, 2001.

[8] J. Chandrashekar, Z. Duan, Z.-L. Zhang, and J. Krasky, "Limiting Path Exploration in Path Vector Protocols," University of Minnesota, Tech. Rep., 2004, http://www.cs.umn.edu/~jaideepc/papers/epic-tr.pdf.

[9] T. Bates, R. Chandra, and E. Chen, "RFC 2796: BGP Route Reflection - An Alternative to Full Mesh IBGP," http://www.ietf.org/rfc/rfc2796.txt, April 2000.

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

[11] A. Bremler-Barr, Y. Afek, and S. Schwarz, "Improved BGP Convergence via Ghost Flushing," in *Proc. IEEE INFOCOM*. IEEE, Mar 2003.

[12] C. Labovitz, A. Ahuja, R. Wattenhofer, and V. Srinivasan, "The Impact of Internet Policy and Topology on Delayed Routing Convergence," in *Proc. IEEE INFOCOM*, 2001, pp. 537–546.

[13] "BRITE: Boston University Representative Internet Topology," http://www.cs.bu.edu/brite/.

[14] S. R. Sangli, D. Tappan, and Y. Rekhter, "BGP extended communities attribute," http://www.ietf.org/internet-drafts/draft-ietf-idr-bgp-ext-communities-%05.txt, Nov 2002, internet Draft.

[15] C. Systems, "Endless bgp convergence problem in cisco ios," October 2000, field Notice.

[16] M. Musuvathi, S. Venkatachary, R. Wattenhofer, C. Labovitz, and A. Ahuja, "BGP-CT: A First Step Towards Fast Internet Fail-Over," Microsoft Research, Tech. Rep., 2000.

[17] D. Pei, M. Azuma, N. Nguyen, J. Chen, D. Massey, and L. Zhang, "BGP-RCN: Improving BGP Convergence through Root Cause Notification," UCLA, Tech. Rep. 030047, October 2003.

[18] H. Zhang, A. Arora, and Z. Liu, "G-BGP: Stable and Fast Convergence of the Border Gateway Protocol," Ohio State University, Tech. Rep. OSU-CISRC-6/03-TR36, June 2003.

[19] D. Pei, X. Zhao, L. Wang, D. Massey, A. Mankin, S. F. Wu, and L. Zhang, "Improving BGP Convergence Through Consistency Assertions," in *Proc. IEEE INFOCOM*. IEEE, 2002.