IP Fast Reroute in Networks with Shared Risk Links

Yan Li
Department of Computer Sciences
The University of Texas at Austin
Austin, Texas 78712
Email: yanli@cs.utexas.edu

Mohamed G. Gouda
The National Science Foundation, and
The University of Texas at Austin
Austin, Texas 78712
Email: gouda@cs.utexas.edu

***Abstract*—IP fast reroute is a mechanism that is used to reroute packets around a failed link, as soon as the link fails and before the employed routing protocol has a chance to adapt the routing tables to the link failure. Most of the IP fast reroute mechanisms, that have been proposed so far, focus on single or dual link failures but can not handle Shared Risk Link Group (SRLG) failures when several links fail at the same time because of some common underlying component failure. Furthermore, most of current work is based on the assumption that each node in the network has access to some global topology information of the network. (This assumption seems reasonable if the employed routing protocol is a link-state protocol, but it is not valid if the employed routing protocol is a distance vector like protocol.) In this paper, we present the first IP fast reroute mechanism for SRLG failures that is not based on the assumption that the nodes in the network have global topology information of the network.**

In our proposed mechanism, any node *x* can advertise that it needs to be able to reroute around a link *x*→*y* that belongs to a SRLG when this link fails. When the nodes in the network receive this advertisement, they start to exchange "relay bits" for link *x*→*y* and some nodes in the network recognize that they can serve as "relay nodes" for link *x*→*y* to avoid any SRLG failures and notify node *x* of this fact. Finally, nodes *x* chooses one of the relay nodes, say node *z*, to be the relay node for link *x*→*y*. Thus, when link *x*→*y* fails, node *x* can immediately tunnel packets that need to traverse link *x*→*y* to node *y* through node *z*. Our tunneling scheme ensures that loops are never formed even when any number of links fail. Through simulation, we show that our mechanism succeeds in rerouting around SRLG failures about 100% of the time, with average length of a reroute path about 1.5 times the re-converged shortest path.

***Keywords*-IP Fast Reroute, Shared Risk Link Group Failure, Distance Vector, Failure Recovery, Reliability**

I. INTRODUCTION

As Internet becomes the ubiquitous infrastructure for various applications and carries all kinds of traffic, traditional best effort service model becomes insufficient. Demands on reliability and availability are becoming more and more stringent, especially with the development of more real-time applications like VoIP and Video on demand [5]. Unfortunately, failures are very common in daily operations of a network and what makes things worse is that most failures are not predictable. It is reported in [21] that 80% of all failures are unexpected. Among these unexpected failures, besides the most common single link failures, another significant part is Shared Risk Link Group (SRLG) failures. Links that belong to the same SRLG share some underlying component either in the optical infrastructure like a fiber or at a router like a line card.

The convergence process for failure recovery in traditional routing protocols, link state and distance vector, is time consuming and may result in instability in case of frequent transient link failures. During this convergence process, the routing tables may be inconsistent and packets may be dropped due to invalid routes. Although much work has been dedicated to reduce the convergence time of routing to even under a second [12], it is still quite far from the 50 milliseconds target for mission critical applications [24].

Recently, IP fast reroute has been proposed to proactively compute backup paths before a failure happens. And as soon as a failure is detected, the backup path can be invoked immediately to reroute around the failure during the convergence period. Thus the routing disruption time can be limited to only the failure detection time [25]. Although several mechanisms have been developed within the IP fast reroute realm, most of them focus on single or dual link failures and can not handle SRLG failures [3], [4], [6], [14], [16], [19], [22], [23]. Also, most of existing work relies on the existence of some global topology information to precompute backup paths [3], [4], [6], [14], [16], [17], [22], [23], [26]. Specifically, for [3], [4], [6], [14], [16], [17], [22], [23], they assume each node in the network has the complete knowledge of all the connectivity information in the network (i.e., how all the nodes in the network are connected with each other). For Not-via [26], each node is allocated some special Not-via IP addresses for all the links associated with that node. These Not-via addresses have to be known and stored in the routing table by every other node in the network, no matter whether the links are intended to be protected or not.

When global topology information is not available (for example, in distance vector like routing protocols), to recover from SRLG failures, IP fast reroute faces more challenges: how to get necessary information to compute alternative backup paths to avoid all the links in the same SRLG, without changing the original routing tables? To address this challenge, we design an IP fast reroute mechanism for handling SRLG failures with the following goals:

- ***No Global Topology Information***: Our mechanism does not assume nodes in the network have access to any global topology information: neither the connectivity information of the network nor any additional IP addresses associated with each node which need to be globally known and stored in routing tables. Each node only has access to its local topology, i.e., links associated with the node itself. Also, each node only knows to which SRLG its associated links belong but does not know other links in the same SRLG.

- ***Distributed Computation of Backup Paths***: Each backup path for a reroute link is designated by a node called *relay node*. In order to find relay nodes in a fully distributed way, we introduce two *relay bits* for each reroute link. Each node only maintain and exchange two relay bits with its neighbors for each reroute link. Using the two relay bits, a node in the network can automatically decide if itself can serve as a relay node for a reroute link or not. Thus backup paths for a reroute link can be computed in a fully distributed manner.

- ***Reroute Only When You Want***: Rerouting information is propagated only for links that are currently under protection using IP fast reroute, which are called *reroute links*. Reroute links can be changed at any time. Thus rerouting information for links that are not necessarily being protected at some instant will not be propagated and stored. So the cost of our mechanism is dynamically associated with the number of links currently protected in the network.

The main idea of our IP fast reroute mechanism to recover from SRLG failures is elaborated as follows. Any node x can decide, at anytime, that it needs to reroute packets around its link $x{\rightarrow}y$ if this link ever fails in the future. When this happens, node x starts advertising that its link $x{\rightarrow}y$, together with its SRLG number, has been designated as a reroute link. When a node z in the network receives this advertisement, node z allocates and maintains two bits for link $x{\rightarrow}y$. The two bits are called the *relay bits* of link $x{\rightarrow}y$ in node z.

The initial value of the relay bits for each reroute link $x{\rightarrow}y$ in each node z is 00. Each node periodically sends its relay bits (for different reroute links in the network) to its neighboring nodes, and uses the received relay bits to update the values of its own relay bits. If the value of the relay bits for a link $x{\rightarrow}y$ in node z ever becomes 11, then node z recognizes that it is a relay node for link $x{\rightarrow}y$ and sends a notification message to the source node x of link $x{\rightarrow}y$.

The source node x receives notification messages from many nodes for link $x{\rightarrow}y$, and selects only one of these nodes, say node z, to be the relay node for link $x{\rightarrow}y$. From this point on, when nodes x receives a packet that needs to be forwarded over link $x{\rightarrow}y$ and discovers that this link failed, node x immediately forwards the packet to the relay node z which then forwards the packet to the other end of the reroute link, node y. This process continues if the packet encounters any other failed links towards its ultimate destination.

We propose a tunneling scheme to ensure that loops are never formed. We also propose an algorithm to suppress redundant relay notification messages. Finally we show, through extensive simulations on a variety of networks of different sizes and varying SRLG size from 1 to 5, that the coverage of our mechanism is close to 100%. Suppression can effectively cut about 80% notification messages when the network has at least one hundred nodes. Also, the average length of a reroute path is around 1.5 times the average length of the re-converged shortest path.

The structure and organization of this paper follows from our technical report [19] which focus on how to handle single link failures using IP fast reroute. Section II presents the concept of reroute links and how to designate reroute links. Then in Section III, we introduce the concept of a relay node for a reroute link that is a member of a SRLG and how to use the relay node in rerouting. Section IV presents how a node learns a relay node, without access to the global topology information, for a reroute link which belongs to a SRLG. Section V describes the suppression mode to suppress redundant notification messages. We show the efficiency and overhead of our rerouting algorithms in Section VI. Related work is reviewed in Section VII. Finally we conclude in Section VIII.

II. Reroute Links in Shared Risk Link Group

We model a network as an undirected graph where each node represents a router and each (undirected) edge between two nodes, say nodes a and b, represents two links: link $a{\rightarrow}b$ and link $b{\rightarrow}a$. For each link $a{\rightarrow}b$, node a is called *the source* of link $a{\rightarrow}b$ and node b is called *the sink* of link $a{\rightarrow}b$.

A Shared Risk Link Group (SRLG) is a set of links which share the same underlying physical point of failure such as fiber cut or line card failure. We assume that all the links in the network are partitioned into different SRLG groups. That is, each link belongs to one SRLG group. If a link does not share physical resources with others, then it is regarded as an SRLG with only one link. We also assume that the source node a of a link $a{\rightarrow}b$ only knows to which shared risk link group link $a{\rightarrow}b$ belongs if any, but node a does not know other links that belong to the same shared risk link group as link $a{\rightarrow}b$.

We assume that packets are routed between different nodes in the network using distance vector routing protocols [2], [20]. In distance vector routing protocols, the periodic exchange of routing tables between neighboring nodes in a network eventually causes the next hop and distance entries in all routing tables in the network to "converge" to their correct values. This convergence procedure may take even tens of seconds.

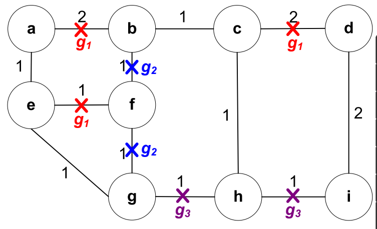

Fig. 1. An example network N_1

Table I
ROUTING TABLE ***RT.a*** OF NODE *a*

dest.	next hop	dist.
a	-	0
b	b	2
c	b	3
d	b	5
e	e	1
f	e	2
g	e	2
h	e	3
i	e	4

As an example, Figure 1 shows a network N_1 that has nine nodes and twelve edges. Each edge in network N_1 is labeled with a distance. The routing table *RT.a* of node *a* in network N_1 is shown in Table I. Also, there are three SRLGs in N_1 with SRLG number g_1, g_2 and g_3. Link *a*→*b*, *c*→*d* and *e*→*f* belong to the same SRLG g_1; link *b*→*f* and *f*→*g* belong to the same SRLG g_2; link *g*→*h* and *h*→*i* belong to the same SRLG g_3.

Now consider the situation where node *a* has a packet whose ultimate destination is node *d*. But then node *a* notices that link *a*→*b* used to reach destination *d* has failed and so no packet can be transmitted over it. The question now is "what does node *a* do with this packet, and every other packet, that need to be transmitted over the failed link *a*→*b* before the entries of the routing table *RT.a* have converged to their new correct values?"

Node *a* has two options in this situation. The first option is that node *a* drops every packet that needs to be transmitted over the failed link *a*→*b*. This option is attractive if link *a*→*b* is very reliable (and so the probability of its failure is very small) or if the rate of packets that need to be transmitted over link *a*→*b* is very small.

The second option is that node *a* anticipates the failure of link *a*→*b* and maintains alternative routes that can be used to reroute around link *a*→*b* when this link fails. To do so, node *a* needs to advertise to every node in the network that link *a*→*b* has been designated (by node *a*) to be a *reroute link*. Thus every node in the network can proceed to help node *a* identify and maintain alternative routes that can be used to reroute around link *a*→*b* when it fails.

Each node *a* in the network is provided with a *rerouting table RR.a* that has the following four columns:

$$(rlink,\ srlg,\ rbits,\ relay)$$

The first column, *rlink*, in every rerouting table lists all the links that have been designated, by their source nodes, as reroute links. The second column, *srlg*, lists the shared risk link group that this reroute link belongs to. The other two columns, *rbits* and *relay*, are discussed below in Section IV.

Initially, the rerouting table *RR.a* of each node *a* is empty except that node *a* adds one entry for each link $a{\to}s_n$ that *a* wants to designate as a reroute link. Whenever node *a* sends a copy of its routing table *RT.a* to each neighboring node, node *a* also sends a copy of its rerouting table *RR.a* (excluding the "relay" column) to the neighboring node. This periodic exchange of routing and rerouting tables between neighboring nodes in the network eventually causes every link that has been designated as a reroute link to have one entry in every rerouting table in the network.

At any time, each node *a* can change the set of links that it has designated as reroute links by adding new links to this set or by removing old links from this set. The algorithm for updating this set is shown in Algorithm 1. Note that in this algorithm, we use *rlink.a* to denote the column *rlink* in the rerouting table *RR.a* of node *a*.

Algorithm 1: Node *a* updates its set of reroute links

```
begin
    for $a{\to}s_n \notin rlink.a$ do
        if a needs to be able to reroute around $a{\to}s_n$ then
            add $a{\to}s_n$ entry to RR.a;
        end
    end
    for $a{\to}s_n \in rlink.a$ do
        if a no longer needs to be able to reroute around
        $a{\to}s_n$ then
            remove $a{\to}s_n$ from RR.a;
        end
    end
end
```

Note that links that belong to the same SRLG can be designated as reroute links seperately depending on the two scenarios we just discussed above.

III. Relay Nodes for SRLG Failures

In this section, we introduce the concept of a relay node for a reroute link that is a member of a SRLG in a network. (Note that the relay node defined in this section also works for reroute links that does not belong to any SRLG in the network.) We then discuss how a relay node for a reroute link can be used in rerouting around its reroute link when all the links that belong to the same SRLG fail.

Since the set of links belonging to the same SRLG shares the same underlying physical point of failure such as fiber cut or line card failure, when repairing the failure of a reroute link that belongs to a SRLG, all the other links that are members of the same SRLG should be avoided since they all fail at the same time.

Let *s* and *d* be two nodes in a network *N*, and let *R(s, d)* denote the shortest route from node *s* to node *d* as determined by the routing tables in network *N*. A node *r* in *N* is called a *relay node* for a reroute link *a*→*b* in *N* iff neither the route *R(a, r)* nor the route *R(r, b)* contains any link, if any, which belongs to the same SRLG as link *a*→*b* (including *a*→*b* itself).

As an example, consider network N_1 in Figure 1. Link *a*→*b*, link *c*→*d* and link *e*→*f* belong to the same SRLG g_1. If link *a*→*b* in network N_1 is designated, by node *a*, as a reroute link, then by definition, node *g*, *h* and *i* are relay nodes for link *a*→*b*. However, if link *b*→*c* and link *c*→*h* belong to the same SRLG (not shown in N_1), and link *b*→*c* is designated, by node *b*, as a reroute link, then no node in network N_1 is a relay node for link *b*→*c* according to the definition of relay nodes. These two examples simply show that a reroute link in a SRLG can have one or more relay nodes or even no relay nodes. Fortunately, we show by extensive simulations below in Section VI that reroute links in a SRLG that have no relay nodes are extremely rare.

Next we describe the procedure for rerouting around a reroute link *a*→*b*, when this link fails, assuming that node *a* knows the identity of a relay node *r* for link *a*→*b*:

1. Assume that a packet is to be sent from a node *s* to a node *d* along the route *R(s, d)* which contains the reroute link *a*→*b*. In this case, the IP header of the packet can be represented as *(from s, to d)*.
2. Assume also that when this packet reaches node *a*, node *a* discovers that the reroute link *a*→*b* has failed and so decides to reroute the packet towards the relay node *r* for link *a*→*b*. In this case, node *a* encapsulates the packet in two outer IP hearders *(from a, to b)* and then *(from a, to r)* and forwards the encapsulated packet towards *r*.
3. When the encapsulated packet reaches the relay node *r*, node *r* removes the outermost IP header *(from a, to r)* and discovers that the packet has an inner IP header *(from a, to b)*. Thus node *r* forwards the encapsulated packet towards *b*.
4. When the encapsulated packet reaches node *b*, node *b* removes the outer IP header *(from a, to b)* and discovers that the packet has an inner IP header *(from s, to d)*, which indicates that the ultimate destination of the packet is *d*. Thus node *b* forwards the packet towards *d*. Note that now the packet is not encapsulated any more.
5. Finally when the packet reaches node *d*, node *d* discovers that the original source of the packet is *s* and its ultimate destination is *d*. Thus, node *d* keeps the packet.
6. Assume that, while the packet is traversing the route *R(b, d)* (non-encapsulated now like a normal packet), the packet reaches a node *x* that needs to transmit the packet over the reroute link *x*→*y* except that it discovers that link *x*→*y* has failed. Node *x* can use the same procedure, step 1 through 5 as describe above, to reroute the packet around the failed link *x*→*y*, no matter whether link *x*→*y* belongs to the same SRLG as link *a*→*b* or not.
7. However, assume that the encapsulated packet is traversing the route *R(a, r)* or the route *R(r, b)*, the packet reaches a link *x*→*y* that has failed. Recognizing that the packet is being rerouted because it is an encapsulated packet, node *x* drops the packet and does not attempt to reroute it a second time. This action guarantees that the routing loops are not created due to repeated rerouting of the same packet. Note that this action is different from step 6 above. In step 6, the packet can be rerouted because it is not in the procedure of rerouting (i.e., the packet is not encapsulated). In this step, no more rerouting should be allowed while the packet is right in the rerouting procedure (i.e., the packet is encapsulated).

The fact that no routing loops are created due to the repeated rerouting of the same packet is established in the following lemma and theorem.

Lemma 1.
Let *r* be a relay node for a reroute link *a*→*b* in a network *N*, and let *d* be any node in network *N* such that the route *R(a, d)* contains the reroute link *a*→*b*. If link *a*→*b* fails, and node *a* reroutes a packet, whose ultimate destination is node *d*, to node *r*, then this packet will not traverse any loop in the network before reaching node *b* no matter whether link *a*→*b* belongs to a SRLG or not.

Proof: First, if when the rerouted packet traverses the route *R(a, r)* or the route *R(r, b)*, the packet does not encounter any other link failures, then from the definition of a relay node, this packet will not traverse the reroute link *a*→*b* and thus will not be rerouted again. Hence the rerouted packet will not traverse any loop before reaching node *b*. Second, if when the rerouted packet traverses the route *R(a, r)* or the route *R(r, b)*, the packet encounters any other link failure, denoted *x*→*y*, then if *x*→*y* is a reroute link, then according to the seventh step of reroute procedure presented above, node *x* checks that the packet is an encapsulated packet (means that it has been rerouted once), *x* just drops the packet to avoid rerouting it again. Thus, the packet is only rerouted once through the relay node *r* and it will not traverse any loop before reaching the sink node *b*. On the other hand, if link *x*→*y* is not a reroute link, node *x* will have no relay node to reroute the packet, so *x* will drop the packet and thus the packet will not traverse any loop before reaching the sink node *b*. ■

Theorem 2.

No routing loops are created due to repeated rerouting of the same packet to its ultimate destination using the rerouting procedure, no matter whether all encourtered failed links belong to the same SRLG or not.

Proof: From Lemma 1, we know that rerouting a packet through a single failed link to the sink node will not create loops. Assume that after a packet is successfully rerouted through a failed link *a*→*b*, when the packet traverses the route *R(b, d)* towards its ultimate destination *d*, it encounters another link failure, denoted *x*→*y*. If link *x*→*y* belongs to the same SRLG as link *a*→*b*, and the relay node for *x*→*y* is denoted *r'*, then according to the definition of relay node, route *R(x, r')* and *R(r', y)* contains neither *a*→*b* nor *x*→*y*. Thus no loops are created due to rerouting the packet again through link *x*→*y*. Otherwise, if link *x*→*y* does not belong to the same SRLG as link *a*→*b*, if either route *R(x, r')* or *R(r', y)* contains *a*→*b*, then after the packet being rerouted through *a*→*b*, it would traverse the link *x*→*y* again (after going through link *a*→*b*). This contradicts to the definition of relay node since *R(x, r')* or *R(r', y)* contains *x*→*y*. Thus, no matter how many failed links encountered when using the rerouting procedure to reroute a packet towards its ultimate destination, and no matter whether or not all these failed links belong to the same SRLG or not, there are no loops created due to rerouting. ■

IV. Relay Bits to identify Relay Nodes

In the previous section we presented a procedure by which a node *a* can reroute packets around a reroute link *a*→*b* when all the links that belong to the same SRLG fail. This procedure is based on the assumption that node *a* knows a relay node for the reroute link *a*→*b*. So the question now is "How does node *a* know a relay node for link *a*→*b* which belongs to a SRLG without access to the global topology information?" In this section we present a fully distributed procedure by which node *a* learns all the relay nodes for link *a*→*b* although node *a* does not know any other links that belong to the same SRLG as link *a*→*b*. This procedure consists of the following three parts.

1. In the first part, node *a* informs every node in the network that it has designated link *a*→*b* as a reroute link as well as the SRLG number g_1 that *a*→*b* belongs to.
2. In the second part, each node *x* in the network receives the information that link *a*→*b* has been designated as a reroute link and checks whether the route *R(x, b)* and *R(x, a)* include any link that belongs to the same SRLG g_1 as link *a*→*b*. If node *x* finds that neither *R(x, b)* nor *R(x, a)* includes any link that belongs to the same SRLG g_1 as link *a*→*b*, then node *x* sends a *notify(x, a*→*b)* message to node *a* to inform *a* that *x* is a *relay node* for link *a*→*b*.
3. In the thrid part, when node *a* receives a *notify(x, a*→*b)* message, node *a* concludes that *x* is a relay node for link *a*→*b*. Node *a* then adds *x* to the set of *relay.a[a*→*b]* in the rerouting table *RR.a* of node *a*.

Next we describe first two parts of the procedure in some details.

The First Part: For node *a* to announce that it has designated link *a*→*b* as a reroute link, node *a* adds the entry (*a*→*b*, g_1, 00, -) to its rerouting table *RR.a*. Recall that each entry in a rerouting table consists of four components *(rlink, srlg, rbits, relay)*, where *rlink* is a reroute link; *srlg* lists the shared risk link group that this link belongs to; *rbits* are two relay bits (to be discussed shortly) for the reroute link; and *relay* is the set of all known relay nodes for the reroute link. The initial value of *relay* is "-" which indicates that node *a* does not know yet this value.

Because the rerouting table of every node is sent periodically to every neighbor of this node, the fact that link *a*→*b* has been designated a reroute link, as well as the SRLG g_1 it blongs to, is eventually recorded in every rerouting table in the network according to the following rule. If a node *x* receives a rerouting table *RR.y* from a neighbor *y*, and the next hop for reaching node *a* in the routing table *RT.x* of node *x* is node *y*, and if *RR.y* has an *a*→*b* entry but *RR.x* does not have *a*→*b* entry, then node *x* adds an entry (*a*→*b*, g_1, 00, -) to its rerouting table *RR.x*. Conversely, if *RR.y* has no *a*→*b* entry but *RR.x* has an *a*→*b* entry, then node *x* removes the *a*→*b* entry from its rerouting table *RR.x*.

The Second Part: For each reroute link *a*→*b* in the rerouting table *RR.x* of each node *x* in the network, node *x* maintains two bits, named the relay bits of link *a*→*b*, in *RR.x*. These two bits are denoted *rbits.x[a*→*b]* and each of the two bits has anyone of two values. The value "0" in the first bit indicates two cases: either node *x* does not know yet the correct value of the bit (i.e., initial value of the bit), or node *x* has checked that some link that belongs to the same SRLG g_1 as link *a*→*b* occurs in the route *R(x, a)*. The value "1" in the first bit indicates that *x* has checked that no link that belongs to the same SRLG g_1 as link *a*→*b* occurs in the route *R(x, a)*. Similarly, the value in the second bit indicates the same meaning except that node *x* checks whether there is any link that belongs to the same SRLG g_1 as link *a*→*b* occurs in the route *R(x, b)*. Only when the two bits are both "1"s, i.e., no link that belongs to the same SRLG g_1 as link *a*→*b* occurs in route *R(x, a)* and route *R(x, b)*, node *x* is a relay node for reroute link *a*→*b*.

Next, we describe how to set up the two relay bits for a reroute link. Initially, the value of *rbits.x[a*→*b]* is "00" in the rerouting table *RR.x* in every node *x* in the network, meaning that every node does not know the correct value of the bits yet. The source node *a* of link *a*→*b* assigns the relay bits *rbits.a[a*→*b]* in its rerouting table *RR.a* the value 10. The first bit "1" means that no link that belongs to the same SRLG g_1 as link *a*→*b* occurs in the route *R(a, a)*, and the second bit "0" means that link *a*→*b* occurs in the route *R(a, b)*.

Then the sink node *b* of link *a→b* assigns the bits *rbits.b[a→b]* in its rerouting table *RR.b* the value 01. The first bit "0" means that link *b→a* that belongs to the same SRLG g_1 as link *a→b* occurs in the route *R(b, a)*, and the second bit "1" means that no link that belongs to the same SRLG g_1 as link *a→b* occurs in the route *R(b, b)*.

Then every other node *x* in the network assigns each of the two relay bits *rbits.x[a→b]* in its rerouting table *RR.x* the value *val*, where *val* is either 0 or 1, according to the following rule: If *x* receives *RR.y* from neighbor *y*, and if the next hop for reaching node *a* in the routing table *RT.x* of node *x* is node *y*, then node *x* checks whether the shared risk link group of link *x→y* is the same as the shared risk link group of link *a→b*, if yes, then node *x* assigns the first bit *rbits.x[a→b][0]* in its *RR.x* the value 0; if no, then node *x* assigns the first bit *rbits.x[a→b][0]* in its *RR.x* the value of the first bit in rbits.y[a→b]. Similarly, node *x* assigns the second relay bit: If the next hop for reaching node *b* in the routing table *RT.x* of node *x* is node *y*, then node *x* checks whether the shared risk link group of link *x→y* is the same as the shared risk link group of link *a→b*, if yes, then node *x* assigns the second bit *rbits.x[a→b][1]* in its *RR.x* the value 0; if no, then node *x* assigns the second bit *rbits.x[a→b][1]* in its *RR.x* the value of the second bit in rbits.y[a→b].

The first and second parts outlined above are part of updating the rerouting table *RR.x* after node *x* receives the rerouting table *RR.y* from the neighboring node *y* shown in Algorithm 2.

Figure 1 shows three SRLGs: g_1, g_2 and g_3. Assume that all the links in three SRLGs g_1, g_2 and g_3 have been designated by their respective source node to be reroute links. Also, assume that link *a→e* has been designated by node *a* as a reroute link and link *d→i* has been designated by node *d* as a reroute link too. But these two links do not belong to any SRLG. Then the rerouting table *RR.a* of node *a*, after these links have been designated as reroute links, is shown in Table II.

Note that *RR.a* includes the relay nodes for the two reroute links *a→b* and *a→g* for which node *a* is the source node. But *RR.a* does not include any relay node for any reroute link for which node *a* is not the source node.

Correctness of the procedure for updating the relay bits follows from the next theorem.

Theorem 3.
For any node *x* in a network N, if the relay bits in node *x* for a reroute link *a→b* are both ones, i.e., *rbits.x[a→b]=11*, then neither route *R(a,x)* nor route *R(x,b)* contains any link that belongs to the same SRLG as reroute link *a→b*.

Proof: We prove the two parts respectively: first, we prove that if the first relay bit, denoted *rbits.x[a→b][0]* is 1, then route *R(a,x)* does not contain any link that belong to the same SRLG as reroute link *a→b*; second, we prove that if the second relay bit, denoted *rbits.x[a→b][1]* is 1, then route *R(x,b)* does not contain any link that belong to the same SRLG as reroute link *a→b*. We prove the first part using induction on the number of hops in a shortest route. The base case is that *rbits.a[a→b][0] = 1* and *R(a, a)* does not contain any link that belong to the same SRLG as link *a→b*. Assume that this proposition holds for node *y* which

Algorithm 2: Update rerouting table ***RR.x*** after *x* receives rerouting table ***RR.y*** from neighbor *y*

```
begin
    for ($s_r \to s_n \in$ rlink.x) and ($s_r \to s_n \notin$ rlink.y) do
        if nexthop.x[$s_r$] == y then
            remove $s_r \to s_n$ entry from RR.x;
        end
    end
    for ($s_r \to s_n \notin$ rlink.x) and ($s_r \to s_n \in$ rlink.y) do
        if nexthop.x[$s_r$] == y then
            add $s_r \to s_n$ entry to RR.x;
        end
    end
    for $s_r \to s_n \in$ rlink.x do
        if x == $s_r$ then
            rbits.x[$s_r \to s_n$] := 10;
        else if x == $s_n$ then
            rbits.x[$s_r \to s_n$] := 01;
        else
            if nexthop.x[$s_r$] == y then
                if srlg.x[$x \to y$] == srlg.x[$s_r \to s_n$] then
                    rbits.x[$s_r \to s_n$][0] := 0;
                else
                    rbits.x[$s_r \to s_n$][0] := rbits.y[$s_r \to s_n$][0];
                end
            end
            if nexthop.x[$s_n$] == y then
                if srlg.x[$x \to y$] == srlg.x[$s_r \to s_n$] then
                    rbits.x[$s_r \to s_n$][1] := 0;
                else
                    rbits.x[$s_r \to s_n$][1] := rbits.y[$s_r \to s_n$][1];
                end
            end
        end
    end
end
```

Table II
REROUTING TABLE ***RR.a*** OF NODE *a* IN NETWORK N_1 WITH THE RELAY NODES FOR THE REROUTE LINKS *a→b* WHOSE SOURCE NODE IS *a*

rlink	srlg	rbits	relay
a→b	g_1	10	g, h, i
a→e	-	10	b, c, d
c→d	g_1	00	-
e→f	g_1	10	-
b→f	g_2	11	-
f→g	g_2	11	-
g→h	g_3	10	-
h→i	g_3	10	-
d→i	-	11	-

is *n*, $n \geq 0$ hops away from the source node *a* of the reroute link *a*→*b*, i.e., *rbits.y[a→b][0] = 1* and route *R(a, y)* does not contain any link that belongs to the same SRLG as link *a*→*b*. Then from line 18-24 in Algorithm 2, another node *x*, which is *n+1* hops away from the source node *a*, sets its first relay bit *rbits.x[a→b][0]* to one, only when its nexthop *y* also has one for *rbits.y[a→b][0]* and link *x*→*y* does not belong to the same SRLG as link *a*→*b* (neither does link *y*→*x* since the topology is symmetric). Thus node *x* only set its first relay bit *rbits.x[a→b][0]* to one when neither *R(a,y)* nor *y*→*x* contains any link that belong to the same SRLG as link *a*→*b*. Equivalently, *rbits.x[a→b][0]* is set to 1 only when route *R(a,x)* does not contain any link that belong to the same SRLG as reroute link *a*→*b*. Thus, we proved the first part. The second part can be proved similarly. ■

V. Suppression Mode

There is one problem concerning the second and third part of the procedure discussed in the previous section: for some reroute links many nodes in the network qualify to be relay nodes and so these many nodes start to send notify messages to the source node of the link and the source node of a reroute link has to process all the notify messages even though the source node needs only one relay node for the link in order to be able to reroute packets around the link when it fails.

As an example, consider the reroute link *a*→*b* in network N_1 in Figure 1. In network N_1, each of the nodes *g*, *h* and *i* qualifies as a relay node for link *a*→*b*. Thus, each of these nodes sends a notification message to node *a*. However node *a* needs only one relay node for link *a*→*b* so that node *a* can reroute packets around link *a*→*b* when this link fails.

In order to minimize the notification messages sent in the network, we introduce a *suppression mode* for the second part of the procedure discussed in the previous section. In the suppression mode, when the relay bits *rbits.x[a→b]* in the rerouting table *RR.x* of node *x* have the value 11, node *x* recognizes that it is a relay node for link *a*→*b* and so it sends a *notify(x, a→b)* message to its next hop for reaching node *a*, which either drops the message (as explained below) or forwards the message to its next hop for reaching node *a*. Thus, if this notify message is not dropped along the way, then this message will travel along the route *R(x, a)* from node *x* to node *a*. If the *notify(x, a→b)* message reaches, along this route, a node *y* where the relay bits *rbits.y[a→b]* in the rerouting table *RR.y* have the value 11, then node *y* drops the *notify(x, a→b)* message knowing that its own *notify(y, a→b)* message is sufficient for node *a* to have one relay node for link *a*→*b*.

If the suppression mode is used in network N_1 in Figure 1, then for reroute link *a*→*b*,

the notify message from node *i* is dropped by node *h*
the notify messages from *h* is dropped by *g*

Thus, node *a* ends up receiving notify messages concerning link *a*→*b* from only node *g* instead of all the three relay nodes: *g*, *h* and i.

The actions of a node *x* concerning the sending and receiving of notify messages are shown in Algorithm 3.

Algorithm 3: Actions of node *x* on sending and receiving relay notify messages

```
/* ---------sending action----------   */
for $s_r \to s_n \in$ rlink.x do
    if (rbits.x[$s_r \to s_n$] == 11) then
        send notify(x, $s_r \to s_n$) to nexthop.x[$s_r$];
    end
end
/* ---------receiving action---------   */
rcv notify(z, $s_r \to s_n$) from a neighbor y do:
    if x == $s_r$ then
        add z to relay.x[$s_r \to s_n$]
    else if (rbits.x[$s_r \to s_n$] == 11) then
        suppress notify(z, $s_r \to s_n$)
    else
        forward notify(z, $s_r \to s_n$) to nexthop.x[$s_r$]
    end
```

VI. Simulation Results

We now evaluate the performance of our IP fast reroute mechanism for various size of shared risk link groups (i.e., the number of links that are members of a SRLG) using simulations. Through simulation, we intend to answer the following questions: 1) What is the repair coverage for various size of SRLGs? 2) what is the efficiency of suppression under different size of SRLGs? 3) What is the chance that a node can have multiple relay nodes to choose for various size of SRLGs? Will the suppression affect this? 4) What is the overhead of using a relay path, which may include several relays for links in the same SRLG, instead of using the re-converged shortest path? How does the suppression affect this?

We conduct our simulations using two general networks, generated using the BRITE tool [1]. The first network satisfies the power law distribution based on the Barabasi-Albert model with parameter *m* = 2. The second is a random network based on the Waxman model with parameters α = 0.15 and β = 0.2. We have experimented with a variety of network size from tens of nodes to hundreds of nodes, with both types of topologies. For each toplogy with E edges, we randomly select S edges, $1 \leq S \leq 5$ that are close to each other to form a SRLG (the number of hops between the first selected edge and any other selected edge is no larger than 0.6 the maximum number of hops in the network). For S = 1, we count every single link failure. For S $>$ 1, we generate up to 1000 different SRLG failures and make sure each SRLG is not a cut of the topology graph.

Let *repair coverage* be the percentage of source-destination pairs in which, when the link *source*→*sink* of a SRLG used to traverse packets from the source to the destination fails, the source can reroute around any failed link in the same SRLG which appears along the path to reach the destination. To compute repair coverage for SRLG link failures, for every source s_r in a source-destination pair $s_r \rightarrow d$, we iteratively mark all the links that belong to the same SRLG as failed and compute all the relay nodes for that failed link. Then we compute the percentage of source-destination pairs $s_r \rightarrow d$ in which the source s_r can find a relay node to reach that destination d. And obviously when the link is not used from the source to any destination, the case is not counted.

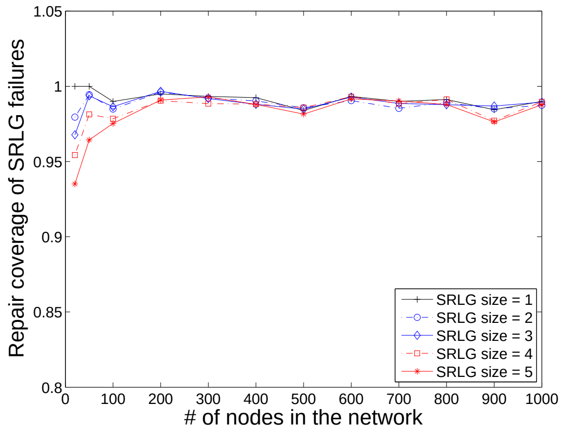

Figure 1. Repair coverage of SRLG failures for Barabasi-Albert networks

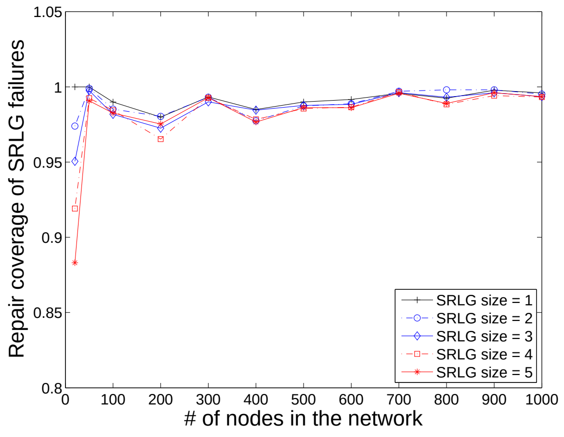

Figure 2. Repair coverage of SRLG failures for Waxman networks

Figure 1 and 2 show the repair coverage for SRLG failures for Barabasi-Albert and Waxman network respectively. For both Barabasi-Albert and Waxman network, no matter what is the size of the network, the repair coverage for smaller SRLG size is greater than the repair coverage for larger SRLG size. However, when the network size is at least 100 nodes, the SRLG size does not have much effect on the repair coverage and our IP fast reroute mechanism can achieve close to 100% repair coverage in these cases.

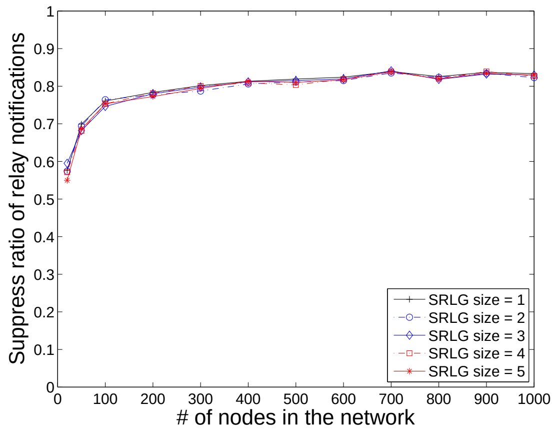

Figure 3. Suppress ratio of relay notify messages for Barabasi-Albert networks

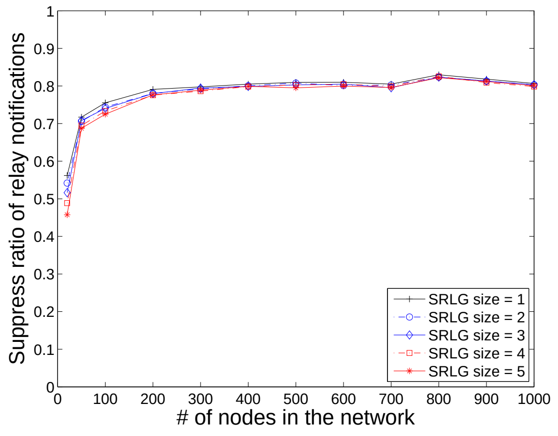

Figure 4. Suppress ratio of relay notify messages for Waxman networks

We measure the efficiency of suppression using *suppress ratio*, which is defined as the percentage of suppressed relay notify messages. As shown in Figure 3 and 4, in both Barabasi-Albert and Waxman networks, the size of the SRLG does not affect the suppress ratio much. And when the network size is at most one hundred nodes, the suppress ratio is between 50% and 75%. If the network size is larger than one hundred nodes, then the suppress ratio is about 80%. This demonstrates that suppression will effectively save the processing overhead for the source node and the bandwidth in the network.

In both Barabasi-Albert and Waxman networks, no matter

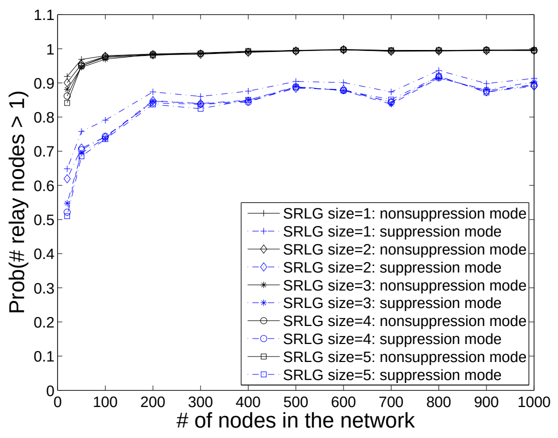

Figure 5. Probability of more than one relay node for SRLG failures in Barabasi-Albert networks

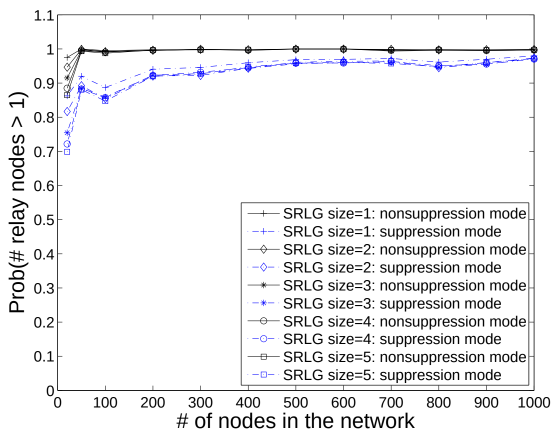

Figure 6. Probability of more than one relay node for SRLG failures in Waxman networks

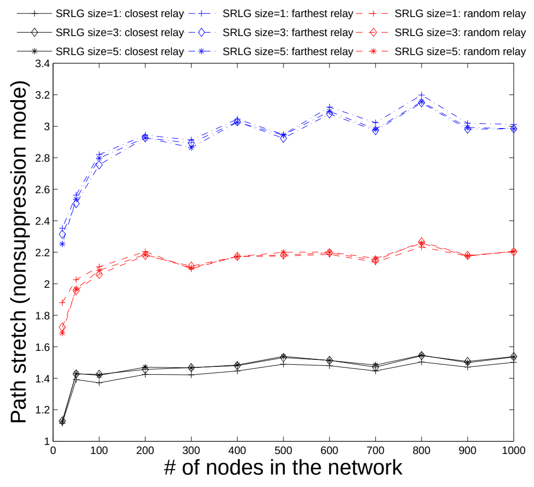

Figure 7. The average path stretch when choosing different relay nodes for SRLG failures in nonsuppression mode for Barabasi-Albert networks

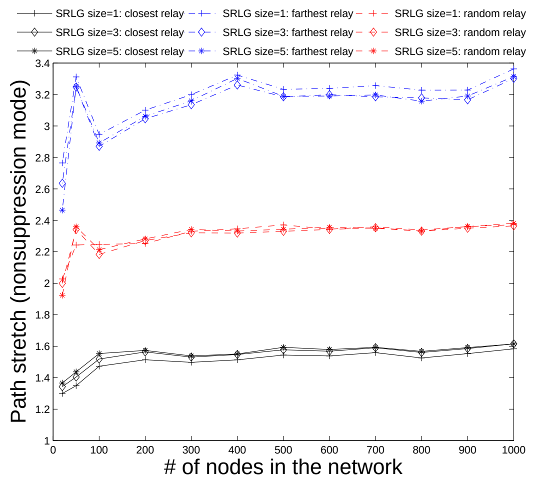

Figure 8. The average path stretch when choosing different relay nodes for SRLG failures in nonsuppression mode for Waxman networks

what's the size for the SRLG, when the network size is over one hundred nodes and there is no suppression, the chance that a source node can find multiple relay nodes to choose from instead of only one relay node is over 97%, shown in Figure 5 and 6. While in the suppression mode, since some relay notify messages are suppressed, the chance that a source node can find multiple relay nodes drops to over 88% in Waxman networks and to about 80% in Barabasi-Albert networks. However, we will show that the suppression mode will not affect the best relay node in terms of reroute path length and it also gives a source node better choices in terms of reroute path length.

For a reroute link, the pre-computed alternative path through a relay node is not necessarily the shortest path. This is because only the source node of the reroute link is aware of the failure and no other nodes are. So compared to the globally re-converged shortest path (which requires the convergence time for the failure to propagate throughout the network), IP fast reroute gains the lossless forwarding with a possible longer path penalty. However, we show that the penalty is not significant. Let *path stretch* be the ratio of the length of the pre-computed alternative path going through the relay node(s) divided by the length of the shorted path after re-convergence. When a source node finds that there are multiple relay nodes for a reroute link, which relay node should the source choose? We examine three choices in terms of path stretch: the closest relay node to the source, the farthest relay node to the source and a random relay node.

In nonsuppression mode (i.e., suppression is not applied), the average path stretch when choosing difference relay nodes for different size of SRLG failures, in Barabasi-Albert and Waxman networks is shown in Figure 7 and 8 respectively. In both networks, no matter what's the size of the SRLG failures, choosing the closest relay node gives the smallest path stretch, less than 1.6 compared to the re-converged shortest path length, while choosing the farthest relay node gives the largest path stretch. A random relay has the stretch in between the above two.

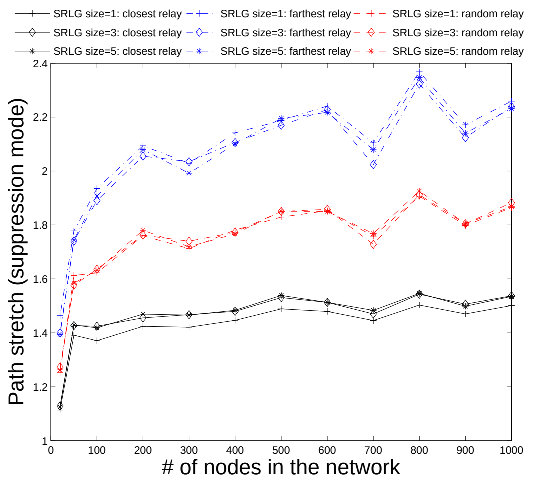

Figure 9. The average path stretch when choosing different relay nodes for SRLG failures in suppression mode for Barabasi-Albert networks

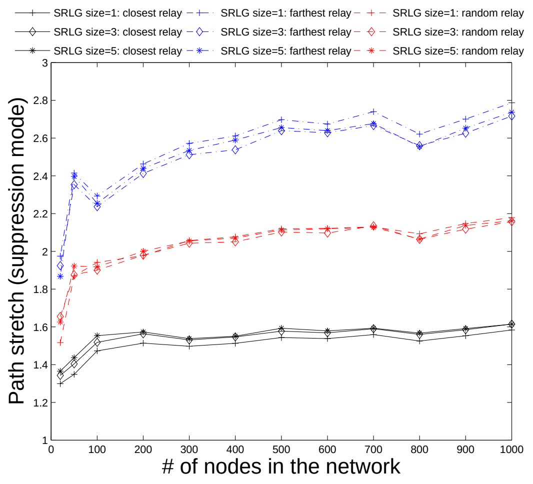

Figure 10. The average path stretch when choosing different relay nodes for SRLG failures in suppression mode for Waxman networks

Figure 9 and 10 show the corresponding path stretch under the suppression mode. It is clear that suppression will not affect the path stretch for closest relay nodes. However, since suppression filters some farther relay nodes which tend to have larger stretch, the average path stretch for both farthest relay nodes and the random relay nodes is reduced under suppression mode. So a source can also randomly choose a relay node with stretch lower than or about 2 in both types of networks.

VII. Related Work

Recently, IP Fast Reroute (IPFRR) has been proposed to recover from failures as soon as a failure is detected using IP-based schemes [25]. However, existing proposals, except [26] which requires substantial number of additional IP addresses, mainly focus on how to handle a single link failure or dual-link failures [3], [4], [6], [14], [16], [19], [23]. Also, most of existing proposals assume each node has the knowledge of some global topology information [3], [4], [6], [8], [14], [16]–[18], [22], [23], [26]. Instead, our work focuses on shared risk link group failures and assumes that each node has neither global connectivity information of the network nor additional global IP addresses information associated with each node. The idea of precomputing backup paths is also explored for BGP [15], [22], [24], [27], [28].

The IPFRR framework [25] requires each router to proactively compute an alternative forwarding path that do not use the failed link or node. Thus, when a failure is actually detected, the alternative path is immediately used during the routing convergence process to avoid dropping packets. Once the routing converges on the new topology, normal routing paths are used to forward packets and each router recomputes a new reroute path after the topology changes. An IPFRR scheme should be able to avoid micro-loops [7], [9], [11]. Francois et al. [10] and Gjoka et al. [13] evaluate the coverage of several IPFRR mechanisms.

Both Loop-free Alternates [4] and U-turn Alternates [3] pre-computes an alternate next hop before a single link failure. Since these two mechanisms find alternates only among next hops, the coverage is not high even for single link failures.

Tunnels [6] is more generalized than the above two mechanisms in the sense that it is not limited to only use next hops as tunnel endpoints, which have loop-free paths to the destination. But again it can only handle single link failures and is only designed for link state protocols. Also it requires a significant number of computations of shortest paths since it computes a reroute path for each of the neighbors of the sink node.

In Multiple Routing Configurations (MRC) [16], each router pre-computes a number of topology configurations by removing rerouted links. Failure Insensitive Routing (FIR) [23] exploits interface-specific forwarding. Both MRC and FIR focus on single link failures. Failure-Carrying Packets (FCP) [17] uses the packet header to carry the list of failed links and requires potentially expensive dynamic computation to route that packet, with the goal of convergence-free routing. Path splicing [22] creates multiple routing trees and allows packets to switch paths by inserting a new packet header. It requires that every node computes k shortest path trees and stores k forwarding table entries for each destination. All these mechanisms require every node to have the knowledge of network topology. In [19], Li et al. explored the idea of using relay nodes to achieve IP fast reroute around single link failures based only on local information.

Kini et. al. [14] proposed an approach to handle two simultaneous link failures by assigning three additional addresses to each node. Their approach also requires every node to be aware of the network topology.

In Not-via [26], to reroute around a failed link, a special Not-via addresses has to be allocated for the sink node regarding that link and advertised over the network. Therefore, each node in Not-via needs d additional Not-via addresses for all the links for which it is a source node, where d is the degree of that node. These additional IP addresses have to be globally known, even when a link is currently not intended to be a reroute link. This significantly increases the size of the routing table and consequently lower the efficiency of forwarding even when there is no failures. Recent work from Li et al. [18] try to improve the efficiency of Not-via by aggregation, but it requires special allocation schemes of Not-via addresses. Enyedi et al. [8] try to reduce the number of Not-via addresses but they also assume the knowledge of global connectivity information.

VIII. Concluding Remarks

We have presented an IP fast reroute mechanism for Shared Risk Link Group failures in routing protocols without global topology information. In our mechanism, any node x can advertise that it needs to be able to reroute around a link $x \rightarrow y$ when this link fails. Then we leverage a set of relay nodes, computed in advance of any link failures, to tunnel the reroute packets around each failed link right after the detection of a failure. Each node uses a fully distributed algorithm to decide automatically whether it can serve as a relay node for a reroute link or not to avoid all link failures in the same SRLG. Notify messages are sent to the source of a reroute link from relay nodes. We proposed a suppression mode to greatly reduce the number of notify messages. Moreover, our tunneling scheme ensures that loops are never formed even when any number of links fail.

Through simulations on different topologies, we confirmed that our mechanism can achieve close to 100% repair coverage in different types and various sizes of networks for different SRLG size. The average length of a reroute path is around 1.5 the re-converged optimal paths. As expected, the suppression is quite effective and cut 80% of notify messages in a network of reasonable size ($\geq$100).

Our future work includes migrating our IP fast reroute mechanism to interdomain routing protocols. Using our mechanism, each AS can potentially leverage the existing Internet topology to achieve fast reroute around Shared Risk Link Group Failures, without changing the BGP advertising and decision process.

References

[1] BRITE: Boston univeristy Representative Internet Topology gEnerator. http://www.cs.bu.edu/BRITE/.

[2] Enhanced interior gateway routing protocol. http://www.cisco.com/en/US/tech/tk365/technologies_white_paper09186a0080094cb7.shtml.

[3] A. Atlas. U-turn alternates for IP/LDP fast-reroute. In *Internet Draft, draft-atlas-ip-local-protect-uturn-03.txt*, February 2006.

[4] A. Atlas and A. Zinin. Basic specification for IP fast reroute: Loop-free alternates. In *RFC 5286*, September 2008.

[5] C. Boutremans, G. Iannaccone, and C. Diot. Impact of link failures on VoIP performance. In *NOSSDAV'02*, May 2002.

[6] S. Bryant, C. Filsfils, S. Previdi, and M. Shand. IP fast reroute using tunnels. In *Internet Draft, draft-bryant-ipfrr-tunnels-03*, November 2007.

[7] S. Bryant and M. Shand. A framework for loop-free convergence. In *Internet Draft, draft-ietf-rtgwg-lf-conv-frmwk-03*, October 2008.

[8] G. Enyedi, P. Szilágyi, G. Rétvári, and A. Császár. IP fast reroute: Lightweight not-via without additional addresses. In *IEEE infocom mini-conference*, 2009.

[9] P. Francois and O. Bonaventure. Avoiding transient loops during IGP convergence in IP networks. In *IEEE Infocom*, 2005.

[10] P. Francois and O. Bonaventure. An evaluations of IP-based fast reroute techniques. In *CoNext*, 2005.

[11] P. Francois, O. Bonaventure, M. Shand, S. Bryant, and S. Previdi. Loop-free convergence using oFIB. In *Internet draft, draft-ietf-rtgwg-ordered-fib-02*, February 2008.

[12] P. Francois, C. Filsfils, J. Evans, and O. Bonaventure. Achieving subsecond IGP convergence in large IP networks. In *ACM Sigcomm*, 2005.

[13] M. Gjoka, V. Ram, and X. Yang. Evaluations of IP fast reroute proposals. In *IEEE Comsware*, 2007.

[14] S. Kini, S. Ramasubramanian, A. Kvalbein, and A. Hansen. Fast recovery from dual link failures in IP networks. In *IEEE Infocom*, 2009.

[15] N. Kushman, S. Kandula, D. Katabi, and B. Maggs. R-BGP: Staying connected in a connected world. In *Usenix NSDI*, 2007.

[16] A. Kvalbein, A. F. Hansen, T. Cicic, S. Gjessing, and O. Lysne. Fast IP network recovery using multiple routing configurations. In *IEEE infocom*, 2006.

[17] K. Lakshminarayanan, M. Caesar, M. Rangan, T. Anderson, S. Shenker, and I. Stoica. Achieving convergence-free routing using failure-carrying packets. In *ACM Sigcomm*, 2007.

[18] A. Li, P. Francois, and X. Yang. On improving the efficiency and manageability of NotVia. In *CoNext*, 2007.

[19] Y. Li and M. G. Gouda. IP fast reroute without global topology information. UTCS Technical Report TR-09-34 (also submitted to a conference), Department of Computer Sciences, The University of Texas at Austin, Austin, TX, 2009.

[20] G. Malkin. RFC 2453 - RIP Version 2, November 1998.

[21] A. Markopoulou, G. Iannaccone, S. Bhattacharyya, C.-N. Chuah, and C. Dio. Characterization of failures in an IP backbone. In *IEEE Infocom'04*, Hong Kong, 2004.

[22] M. Motiwala, M. Elmore, N. Feamster, and S. Vempala. Path splicing. In *ACM Sigcomm*, 2008.

[23] S. Nelakuditi, S. Lee, Y. Yu, Z.-L. Zhang, and C.-N. Chuah. Fast local rerouting for handling transient link failures. *IEEE/ACM Transaction on Networking*, 15(2), April 2007.

[24] C. F. Olivier Bonaventure and P. Francois. Achieving sub-50 milliseconds recovery upon BGP peering link failures. *IEEE/ACM Transaction on Networking*, 15(5), October 2007.

[25] M. Shand and S. Bryant. IP fast reroute framework. In *Internet Draft, draft-ietf-rtgwg-ipfrr-framework-08.txt*, February 2008.

[26] M. Shand, S. Bryant, and S. Previdi. IP fast reroute using Not-via addresses. In *draft-ietf-rtgwg-ipfrr-notvia-addresses-03*, October 2008.

[27] F. Wang and L. Gao. A backup route aware routing protocol - fast recovery from transient routing failures. In *IEEE Infocom mini-conference*, 2008.

[28] F. Wang and L. Gao. Path diversity aware interdomain routing. In *IEEE Infocom*, 2009.