Efficient Recovery from False State in Distributed Routing Algorithms

Daniel Gyllstrom, Sudarshan Vasudevan, Jim Kurose, Gerome Miklau
Department of Computer Science
University of Massachusetts Amherst
{dpg, svasu, kurose, miklau}@cs.umass.edu

***Abstract*—Malicious and misconfigured nodes can inject incorrect state into a distributed system, which can then be propagated system-wide as a result of normal network operation. Such false state can degrade the performance of a distributed system or render it unusable. In the case of network routing algorithms, for example, false state corresponding to a node incorrectly declaring a cost of 0 to all destinations (maliciously or due to misconfiguration) can quickly spread through the network, causing other nodes to (incorrectly) route via the misconfigured node, resulting in suboptimal routing and network congestion. We propose three algorithms for efficient recovery in such scenarios and prove the correctness of each of these algorithms. Through simulation, we evaluate our algorithms when applied to removing false state in distance vector routing, in terms of message and time overhead. Our analysis shows that over topologies where link costs remain fixed, a recovery algorithm based on system-wide checkpoints and a rollback mechanism yields superior performance. We find that a different algorithm – one that selectively purges false routing state network-wide – yields the best performance in scenarios where link costs change.**

I. Introduction

Malicious and misconfigured nodes can degrade the performance of a distributed system by injecting incorrect state information. Such false state can then be further propagated through the system either directly in its original form or indirectly, e.g., as a result of diffusing computations initially using this false state. In this paper, we consider the problem of removing such false state from a distributed system.

In order to make the false-state-removal problem concrete, we investigate distance vector routing as an instance of this problem. Distance vector forms the basis for many routing algorithms widely used in the Internet (e.g., BGP, a path-vector algorithm) and in multi-hop wireless networks (e.g., AODV, diffusion routing). However, distance vector is vulnerable to compromised nodes that can potentially flood a network with false routing information, resulting in erroneous least cost paths, packet loss, and congestion. Such scenarios have occurred in practice. For example, in 1997 a significant portion of Internet traffic was routed through a single misconfigured router, rendering a large part of the Internet inoperable for several hours [19]. More recently [1], a routing error forced Google to redirect its traffic through Asia, causing congestion that left many Google services unreachable. Distance vector currently has no mechanism to recover from such scenarios. Instead, human operators are left to manually reconfigure routers. It is in this context that we propose and evaluate automated solutions for recovery.

In this paper, we design, develop, and evaluate three different approaches for correctly recovering from the injection of false routing state (e.g., a compromised node incorrectly claiming a distance of 0 to all destinations). Such false state, in turn, may propagate to other routers through the normal execution of distance vector routing, making this a network-wide problem. Recovery is correct if the routing tables in all nodes have converged to a global state in which all nodes have removed each compromised node as a destination, and no node bears a least cost path to any destination that routes through a compromised node.

Specifically, we develop three novel distributed recovery algorithms: 2nd `best`, `purge`, and `cpr`. 2nd `best` performs localized state invalidation, followed by network-wide recovery. Nodes directly adjacent to a compromised node locally select alternate paths that avoid the compromised node; the traditional distributed distance vector algorithm is then executed to remove remaining false state using these new distance vectors. The `purge` algorithm performs global false state invalidation by using diffusing computations to invalidate distance vector entries (network-wide) that routed through a compromised node. As in 2nd `best`, traditional distance vector routing is then used to recompute distance vectors. `cpr` uses local snapshots and a rollback mechanism to implement recovery. Although our solutions are tailored to distance vector routing, we believe they represent approaches that are applicable to other instances of this problem.

We prove the correctness of each algorithm and evaluate its efficiency in terms of message overhead and convergence time via simulation. Our simulations show that when considering topologies in which link costs remain fixed, `cpr` outperforms both `purge` and 2nd `best` (at the cost of checkpoint memory). This is because `cpr` can efficiently remove all false state by simply rolling back to a checkpoint immediately preceding the injection of false routing state. In scenarios where link costs can change, `purge` outperforms `cpr` and 2nd `best.` `cpr` performs poorly because, following rollback, it must process the valid link cost changes that occurred since the false routing state was injected; 2nd `best` and `purge`, however, can make use of computations subsequent to the injection of false routing state that did not depend on the false routing state. We will see, however, that 2nd `best` performance suffers because of the so-called count-to-∞ problem.

Recovery from false routing state is closely related to the problem of recovering from malicious transactions [15] [4] in distributed databases. Our problem is also similar to that of rollback in optimistic parallel simulation [13]. However, we are unaware of any existing solutions to the problem of recovering from false routing state. A closely related problem to the one considered in this paper is that of discovering misconfigured nodes. In Section II, we discuss existing solutions to this problem. In fact, the output of these algorithms serve as input to the recovery algorithms proposed in this paper.

This paper has six sections. In Section II we define the problem and state our assumptions. We present our three recovery algorithms in Section III. Then, in Section IV, we present a qualitative evaluation of our recovery algorithms. Section V describes our simulation study. We detail related work in Section VI and finally we conclude and comment on directions for future work in Section VII.

II. Problem Formulation

We consider distance vector routing [5] over arbitrary network topologies. We model a network as an undirected graph, $G = (V, E)$, with a link weight function $w : E \to \mathbb{N}$. Each node, v, maintains the following state as part of distance vector: a vector of all adjacent nodes ($adj(v)$), a vector of least cost distances to all nodes in G ($\overrightarrow{min_v}$), and a *distance matrix* that contains distances to every node in the network via each adjacent node ($dmatrix_v$).

We assume that the identity of the compromise node is provided by a different algorithm, and thus do not consider this problem in this paper. Examples of such algorithms include [7], [8], [9] in the context of wired networks and [21] in the wireless setting. Specifically, we assume that at time t, this algorithm is used to notify all neighbors of the compromised node(s) that a node was compromised. Let t' be the time the node was compromised.

For each of our algorithms, the goal is for all nodes to recover "correctly": all nodes should remove the compromised node as a destination and find new least cost distances that do not use the compromised node. If the network becomes disconnected as a result of removing the compromised node, all nodes need only compute new least cost distances to all other nodes within their connected component.

For simplicity, let $\overline{v}$ denote the compromised node, let $\overrightarrow{old}$ refer to $\overrightarrow{min_{\overline{v}}}$ before $\overline{v}$ was compromised, and let $\overrightarrow{bad}$ denote $\overrightarrow{min_{\overline{v}}}$ after $\overline{v}$ has been compromised. Table I summarizes the notation used in this document.

III. Recovery Algorithms

In this section we propose three new recovery algorithms: 2nd `best`, `purge`, and `cpr`. With one exception, the input and output of each algorithm is the same. [1]

Input: Undirected graph, $G = (V, E)$, with weight function $w : E \to \mathbb{N}$. $\forall v \in V$, $\overrightarrow{min_v}$ and $dmatrix_v$ are computed (using distance vector). Also, each $v \in adj(\overline{v})$ is notified that $\overline{v}$ was compromised.

[1]Additionally, as input `cpr` requires that each $v \in adj(\overline{v})$ is notified of the time, t', in which $\overline{v}$ was compromised.

Abbreviation	Meaning
$\overrightarrow{min_i}$	node i's the least cost vector
$dmatrix_i$	node i' distance matrix
Δ_{lc}	link cost change event
t	time the oracle detects the compromised node
t'	time the compromised node was compromised
$\overrightarrow{bad}$	compromised node's least cost vector at and after t
$\overrightarrow{old}$	compromised node's least cost vector at and before t'
$\overline{v}$	compromised node
$adj(v)$	nodes adjacent to v

TABLE I
Table of abbreviations.

Output: Undirected graph, $G' = (V', E')$, where $V' = V - \{\overline{v}\}$, $E' = E - \{(\overline{v}, v_i) \mid v_i \in adj(\overline{v})\}$, and link weight function $w : E \to \mathbb{N}$. $\overrightarrow{min_v}$ and $dmatrix_v$ are computed via the algorithms discussed below $\forall v \in V'$.

First we describe a preprocessing procedure common to all three recovery algorithms. Then we describe each recovery algorithm.

A. Preprocessing

All three recovery algorithms share a common preprocessing procedure. The procedure removes $\overline{v}$ as a destination and finds the node IDs in each connected component. This could be implemented (as we have done here) using diffusing computations [6] initiated at each $v \in adj(\overline{v})$. A diffusing computation is a distributed algorithm started at a source node which grows by sending queries along a spanning tree, constructed simultaneously as the queries propagate through the network. When the computation reaches the leaves of the spanning tree, replies travel back along the tree towards the source causing the tree to shrink. The computation eventually terminates when the source receives replies from each of its children in the tree.

In our case, each diffusing computation message contains a vector of node IDs. When a node receives a diffusing computation message, the node adds its ID to the vector and removes $\overline{v}$ as a destination. At the end of the diffusing computation, each $v \in adj(\overline{v})$ has a vector that includes all nodes in v's connected component. Finally, each $v \in adj(\overline{v})$ broadcasts the vector of node IDs to all nodes in their connected component. In the case where removing $\overline{v}$ partitions the network, each node will only compute shortest paths to nodes in the vector.

Consider the example in Figure 1 where $\overline{v}$ is the compromised node. When i receives the notification that $\overline{v}$ has been compromised, i removes $\overline{v}$ as a destination and then initiates a diffusing computation. i creates a vector and adds its node ID to the vector. i sends a message containing this vector to j and k. Upon receiving i's message, j and k both remove $\overline{v}$ as a destination and add their own ID to the message's vector. Finally, l and d receive a message from j and k, respectively. l and d add their node own ID to the message's vector and remove $\overline{v}$ as a destination. Then, l and d send an ACK message back to j and k, respectively, with the complete list of node IDs. Eventually when i receives the ACKs from j and k, i has a complete list of nodes in its connected component. Finally, i broadcasts the vector of node IDs in its connected component.

B. The 2^{nd} best Algorithm

2^{nd} `best` invalidates state locally and then uses distance vector to implement network-wide recovery. Following the preprocessing described in Section III-A, each neighbor of the compromised node locally invalidates state by selecting the least cost pre-existing alternate path that does not use the compromised node as the first hop. The resulting distance vectors trigger the execution of traditional distance vector to remove the remaining false state. Algorithm 1 in the Appendix gives a complete specification of 2^{nd} `best`.

We trace the execution of 2^{nd} `best` using the example in Figure 1. At time $t' + \Delta$ (Figure 1(b)), i uses $\overline{v}$ to reach nodes l and d. j uses i to reach all nodes except l. Notice that when j uses i to reach d, it transitively uses $\overrightarrow{bad}$ (e.g., uses path $j-i-\overline{v}-d$ to d). After the preprocessing completes, i selects a new neighbor to route through to reach l and d by finding its new smallest distance in $dmatrix_i$ to these destinations: i selects the routes via j to l with a cost of 100 and i picks the route via k to reach d with cost of 100. (No changes are required to route to j and k because i uses its direct link to these two nodes). Then, using traditional distance vector i sends $\overrightarrow{min_i}$ to j and k. When j receives $\overrightarrow{min_i}$, j must modify its distance to d because $\overrightarrow{min_i}$ indicates that i's least cost to d is now 100. j's new distance value to d becomes 150, using the path $j-i-k-l$. j then sends a message sharing $\overrightarrow{min_j}$ with its neighbors. From this point, recovery proceeds according by using traditional distance vector.

2^{nd} `best` is simple and makes no synchronization assumptions. However, 2^{nd} `best` is vulnerable to the count-to-∞ problem. Because each node only has local information, the new shortest paths may continue to use $\overline{v}$. For example, if $w(k, d) = 400$ in Figure 1, a count-to-∞ scenario would arise. After notification of $\overline{v}$'s compromise, i would select the route via j to reach d with cost 151 (by consulting $dmatrix_i$), using a path that does not actually exist in G ($i-j-i-\overline{v}-d$), since j has removed $\overline{v}$ as a neighbor. When i sends $\overrightarrow{min_i}$ to j, j selects the route via i to d with cost 201. Again, the path $j-i-j-i-\overline{v}-d$ does not exist. In the next iteration, i picks the route via j having a cost of 251. This process continues until each node finds their correct least cost to d. We will see in our simulation study that the count-to-∞ problem can incur significant message and time costs.

C. The purge Algorithm

`purge` globally invalidates all false state using a diffusing computation and then uses distance vector to compute new distance values that avoid all invalidated paths. The diffusing computation is initiated at the neighbors of $\overline{v}$ because only these nodes are aware if $\overline{v}$ is used an intermediary node. The diffusing computations spread from $\overline{v}$'s neighbors to the network edge, invalidating false state at each node along the way. Then ACKs travel back from the network edge to the neighbors of $\overline{v}$, indicating that the diffusing computation is complete. See Algorithm 2 and 3 in the Appendix for a complete specification of this diffusing computation. Next, `purge` uses distance vector to recompute least cost paths invalidated by the diffusing computations.

In Figure 1, the diffusing computation executes as follows. First, i sets its distance to l and d to ∞ (thereby invalidating i's path to l and d) because i uses $\overline{v}$ to route these nodes. Then, i sends a message to j and k containing l and d as invalidated destinations. When j receives i's message, j checks if it routes via i to reach l or d. Because j uses i to reach d, j sets its distance estimate to d to ∞. j does not modify its least cost to l because j does not route via i to reach l. Next, j sends a message that includes d as an invalidated destination. l performs the same steps as j. After this point, the diffusing computation ACKs travel back towards i. When i receives an ACK, the diffusing computation is complete. At this point, i needs to compute new least costs to node l and d because i's distance estimates to these destinations are ∞. i uses $dmatrix_i$ to select its new route to l (which is via j) and uses $dmatrix_i$ to find i's new route to d (which is via k). Both new paths have cost 100. Finally, i sends $\overrightarrow{min_i}$ to its neighbors, triggering the execution of distance vector to recompute the remaining distance vectors.

Note that a consequence of the diffusing computation is that not only is all $\overrightarrow{bad}$ state deleted, but all $\overrightarrow{old}$ state as well. Consider the case when $\overline{v}$ is detected before node i receives $\overrightarrow{bad}$. It is possible that i uses $\overrightarrow{old}$ to reach a destination, d. In this case, the diffusing computation will set i's distance to d to ∞.

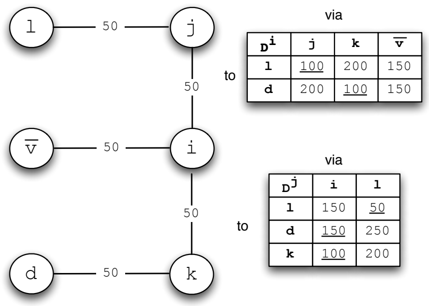
An advantage of `purge` is that it makes no synchronization assumptions. Also, the diffusing computations ensure that the count-to-∞ problem does not occur by removing false state from the entire network. However, globally invalidating false state can be wasteful if valid alternate paths are locally available.

D. The cpr Algorithm

`cpr`[2] is our third and final recovery algorithm. Unlike 2^{nd} `best` and `purge`, `cpr` only requires that clocks across different nodes be loosely synchronized i.e. the maximum clock offset between any two nodes is assumed to be δ. For ease of explanation, we describe `cpr` as if the clocks at different nodes are perfectly synchronized. Extensions to handle loosely synchronized clocks should be clear. Accordingly, we assume that all neighbors of $\overline{v}$, are notified of the time, t', at which $\overline{v}$ was compromised.

For each node, $i \in G$, `cpr` adds a time dimension to $\overrightarrow{min_i}$ and $dmatrix_i$, which `cpr` then uses to locally archive a complete history of values. Once the compromised node is discovered, the archive allows the system to rollback to a system snapshot from a time before $\overline{v}$ was compromised. From this point, `cpr` needs to remove $\overline{v}$ and $\overrightarrow{old}$ and update stale

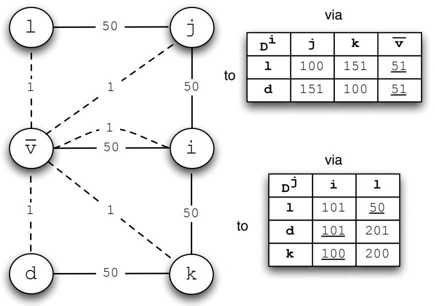
[2]The name is an abbreviation for **C**heck**P**oint and **R**ollback.



D^i	j	k	$\overline{v}$
l	$\underline{100}$	200	150
d	200	$\underline{100}$	150

D^j	i	l
l	150	$\underline{50}$
d	$\underline{150}$	250
k	$\underline{100}$	200

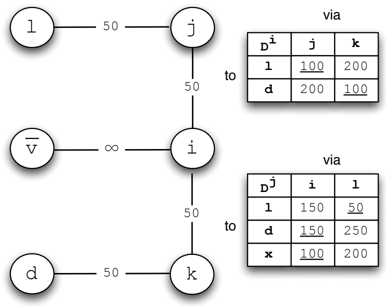
(a) Before t'



D^i	j	k	$\overline{v}$
l	100	151	$\underline{51}$
d	151	100	$\underline{51}$

D^j	i	l
l	101	$\underline{50}$
d	$\underline{101}$	201
k	$\underline{100}$	200

(b) $t' + \Delta$



D^i	j	k
l	$\underline{100}$	200
d	200	$\underline{100}$

D^j	i	l
l	150	$\underline{50}$
d	$\underline{150}$	250
x	$\underline{100}$	200

(c) After recovery

Fig. 1. Three snapshots of a graph, G, where $\overline{v}$ is the compromised node: (a) G before $\overline{v}$ goes bad, (b) G after $\overrightarrow{bad}$ has finished propagating but before recovery has started, and (c) G after recovery. The dashed lines in (b) indicate paths using $\overrightarrow{bad}$. $dmatrix_i$ and $dmatrix_j$, at the time of the snapshot, are displayed to the right of each sub-figure. The least cost values are underlined.

distance values resulting from link cost changes. We describe each algorithm step in detail.

Step 1: Create a $\overrightarrow{min}$ and $dmatrix$ archive. We define a *snapshot* of a data structure to be a copy of all current distance values along with a timestamp. [3] The timestamp marks the time at which that set of distance values start being used. $\overrightarrow{min}$ and $dmatrix$ are the only data structures that need to be archived. This approach is similar to ones used in temporal databases [16], [14].

Our distributed archive algorithm is quite simple. Each node has a choice of archiving at a given frequency (e.g., every m timesteps) or after some number of distance value changes (e.g., each time a distance value changes). Each node must choose the same option, which is specified as an input parameter to `cpr`. A node archives independently of all other nodes. A side effect of independent archiving, is that even with perfectly synchronized clocks, the union of all snapshots may not constitute a globally consistent snapshot. For example, a link cost change event may only have propagated through part of the network, in which case the snapshot for some nodes will reflect this link cost change (i.e., among nodes that have learned of the event) while for other nodes no local snapshot will reflect the occurrence of this event. We will see that a globally consistent snapshot is not required for correctness.

Step 2: Rolling back to a valid snapshot. Rollback is implemented using diffusing computations. Neighbors of the compromised node independently select a snapshot to roll back to, such that the snapshot is the most recent one taken before t'. Each such node, i, rolls back to this snapshot by restoring the $\overrightarrow{min}_i$ and $dmatrix_i$ values from the snapshot. Then, i initiates a diffusing computation to inform all other nodes to do the same. If a node has already rolled back and receives an additional rollback message, it is ignored. (Note that this rollback algorithm ensures that no reinstated distance value uses $\overrightarrow{bad}$ because every node rolls back to a snapshot with a timestamp less that t'.) Algorithm 4 in the Appendix gives the pseudo-code for the rollback algorithm.

Step 3: Steps after rollback. After Step 2 completes, the algorithm in Section III-A is executed. There are two issues to address. First, some nodes may be using $\overrightarrow{old}$. Second, some nodes may have stale state as a result of link cost changes that occurred during $[t', t]$ and consequently are not reflected in the snapshot. To resolve these issues, each neighbor, i, of $\overline{v}$, sets its distance to $\overline{v}$ to ∞ and then selects new least cost values that avoid the compromised node, triggering the execution of distance vector to update the remaining distance vectors. That is, for any destination, d, where i routes via $\overline{v}$ to reach d, i uses $dmatrix_i$ to find a new least cost to d. If a new least costs value is used, i sends a distance vector message to its neighbors. Otherwise, i sends no message. Messages sent trigger the execution of distance vector.

During the execution of distance vector, each node uses the most recent link weights of its adjacent links. Thus, if the same link changes cost multiple times during $[t', t]$, we ignore all changes but the most recent one. Algorithm 5 specifies Step 3 of `cpr`.

In the example from Figure 1, the global state after rolling back is nearly the same as the snapshot depicted in Figure 1(c): the only difference between the actual system state and that depicted in Figure 1(c) is that in the former $(i,\overline{v}) = 50$ rather than ∞. Step 3 in `cpr` makes this change. Because no nodes use $\overrightarrow{old}$, no other changes take place.

Rather than using an iterative process to remove false state (like in 2nd `best` and `purge`), `cpr` does so in one diffusing computation. However, `cpr` incurs storage overhead resulting from periodic snapshots of $\overrightarrow{min}$ and $dmatrix$. Also, after rolling back, stale state may exist if link cost changes occur during $[t', t]$. This can be expensive to update. Finally, unlike `purge` and 2nd `best`, `cpr` requires loosely synchronized clocks because without a bound on the clock offset, nodes may rollback to highly inconsistent local snapshots. Although correct, this would severely degrade `cpr` performance.

[3] In practice, we only archive distance values that have changed. Thus each distance value is associated with its own timestamp.

IV. Analysis of Algorithms

In Section IV-A, we prove the correctness of our three recovery algorithms. Then, we prove specific properties of these recovery algorithms in Section IV-B, which help better understand our simulation results.

A. Correctness of Recovery Algorithms

We make the following assumptions in our proofs. All the initial $dmatrix$ values are nonnegative. Furthermore, all $\overrightarrow{min}$ values periodically exchanged between neighboring nodes are nonnegative. All $v \in V$ know their adjacent link costs. All link weights in G (and therefore G' as well) are nonnegative and do not change. [4] G is finite and connected.

Finally, we assume reliable communication.

Definition 1. An algorithm is *correct* if the following two conditions are satisfied. One, $\forall v \in V$, v has the least cost and knows next-hop to all destinations $v' \in V$. Two, the least cost is computed in finite time.

Theorem 1. *Distance vector is correct.*

Proof. Bertsekas and Gallager [5] prove correctness for distributed Bellman-Ford for arbitrary nonnegative $dmatrix$ values. Their distributed Bellman-Ford algorithm is the same as the distance vector algorithm used in this paper. □

Corollary 1. 2nd `best` *is correct.*

Proof. As per the preprocessing step, each node receiving a diffusing computation message removes $\overline{v}$ as a destination. Each node is guaranteed to receive a diffusing computation message (by our reliable communication and finite graph assumptions). Further, the diffusing computation terminates in finite time. Thus, we conclude that each $v \in V'$ removes $\overline{v}$ in finite time.

Following the diffusing computation, each $v \in adj(\overline{v})$ uses distance vector to determine new least cost paths. Because all $dmatrix_v$ are nonnegative for all $v \in V'$, by Theorem 1 we conclude 2nd `best` is correct. □

Corollary 2. `purge` *is correct.*

Proof. The diffusing computation starts with each $v \in adj(\overline{v})$ finding every destination, d, to which v's least cost path uses $\overline{v}$ as the first-hop node. v sets its least cost to each such d to ∞, thereby invalidating its path to d. v then initiates a diffusing computation. When an arbitrary node, i, receives a diffusing computation message from j, i iterates through each d specified in the message. If i routes via j to reach d, i sets its least cost to d to ∞, therefore invalidating any path to d with j and $\overline{v}$ an intermediate nodes.

By our assumptions, each node receives a diffusing computation message and the diffusing computation terminates in finite time. Thus, we conclude that all paths using $\overline{v}$ as an intermediary node are invalidated in finite time. Following the preprocessing, each $v \in adj(\overline{v})$ uses distance vector to determine new least cost paths. Because all $dmatrix_v$ are nonnegative for all $v \in V'$, by Theorem 1 we conclude that `purge` is correct. □

Corollary 3. `cpr` *is correct.*

Proof. `cpr` rolls back using a diffusing computation. Each node that receives a diffusing computation message, rolls back to a snapshot with timestep less than t'. By our assumptions, all nodes receive a message and the diffusing computation terminates in finite time. Thus, we conclude that each node $v \in V'$ rolls back to a snapshot with timestamp less than t' in finite time.

`cpr` then runs the preprocessing algorithm described in Section III-A, which removes $\overline{v}$ as a destination in finite time (as shown in Corollary 1). Because each node rolls back to a snapshot in which all least costs are nonnegative and `cpr` then uses distance vector to compute new least costs, by Theorem 1 we conclude that `cpr` is correct. □

B. Properties of Recovery Algorithms

In this section we formally characterize how $\overrightarrow{min}$ values change during recovery. The properties established in this section will aid in understanding the simulation results presented in Section V. Our proofs assume that link costs remain fixed during recovery (i.e., during $[t', t]$). We prove properties about $\overrightarrow{min}$ in order provide a precise characterization of recovery trends. In particular, our proofs establish that:

- The least cost between two nodes at the start of recovery is less than or equal to the least cost when recovery has completed. (Theorem 2)
- Before recovery begins, if the least cost between two nodes is less than its cost when recovery is complete, the path must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively. (Corollary 4)
- During 2nd `best` and `cpr` recovery, if the least cost between two nodes is less than its distance when recovery is complete, the path must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively. (Corollary 5)

The first two statements apply to any recovery algorithm because they make no claims about $\overrightarrow{min}$ values during recovery.

Notation. We use the definition of G and G' described in Section III. Let $n, d \in V'$. Let $p_s(n,d)$ be the least cost path from node n to d at the start of recovery and $\delta_s(n,d)$ the cost of this path; $p_i(n,d)$ is a path from n to d used during the recovery and $\delta_i(n,d)$ the cost of this path [5]; and $p_f(n,d)$ the least cost path from n to d when recovery is finished and has cost $\delta_f(n,d)$.

Theorem 2. $\forall n, d \in V'$, $\delta_s(n,d) \leq \delta_f(n,d)$.

Proof: Assume $\exists n_i, d_i \in V'$ such that $\delta_s(n_i,d_i) > \delta_f(n_i,d_i)$. The paths available at the start of recovery are a superset of those available when recovery is complete. This means $p_f(n_i,d_i)$ is available before recovery begins. Distance vector would use this path rather than $p_s(n_i,d_i)$, implying that $\delta_s(n_i,d_i) = \delta_f(n_i,d_i)$, a

[4]We use the definition of G and G' described in Section III.

[5]$p_i(n,d)$ and $\delta_i(n,d)$ can change over time during recovery.

contradiction. □

Corollary 4. *$\forall n, d \in V'$, if $\delta_s(n,d) < \delta_f(n,d)$, then $p_s(n,d)$ is using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively.*

Proof: Assume $\exists n_i, d_i \in V$ such that a path $p_s(n_i, d_i)$ with cost $\delta_s(n_i, d_i)$ is used before recovery begins where $\delta_s(n_i, d_i) < \delta_f(n_i, d_i)$ and $p_s(n_i, d_i)$ does not use $\overrightarrow{bad}$ or $\overrightarrow{old}$. The only paths available before recovery begins, which do not exist when recovery completes, are ones using $\overrightarrow{bad}$ or $\overrightarrow{old}$. Therefore, $p_s(n_i, d_i)$ must be available after recovery completes since we have assumed that $p_s(n_i, d_i)$ does not use $\overrightarrow{bad}$ or $\overrightarrow{old}$. Distance vector would use $p_s(n_i, d_i)$ instead of $p_f(n_i, d_i)$ because $\delta_s(n_i, d_i) < \delta_f(n_i, d_i)$. However this would imply that $\delta_s(n_i, d_i) = \delta_f(n_i, d_i)$, a contradiction. □

Corollary 5. *For* 2nd `best` *and* `cpr`. *$\forall n, d \in V'$, if $\delta_i(n,d) < \delta_f(n,d)$ then $p_i(n,d)$ must be using $\overrightarrow{bad}$ or $\overrightarrow{old}$ either directly or transitively* [6]

Proof: We can use the same proof for Corollary 4 if we substitute $\delta_i(n,d)$ for $\delta_s(n,d)$ and $p_i(n,d)$ for $p_s(n,d)$. □

V. Evaluation

In this section, we use simulations to characterize the performance of each of our three recovery algorithms in terms of message and time overhead. Our goal is to illustrate the relative performance of our recovery algorithms over different topology types (e.g., Erdös-Rényi graphs, Internet-like graphs) and different network conditions (e.g., fixed link costs, changing link costs). We evaluate recovery after a single compromised node has distributed false routing state.

We build a custom simulator with a synchronous communication model: nodes send and receive messages at fixed epochs. In each epoch, a node receives a message from all its neighbors and performs its local computation. In the next epoch, the node sends a message (if needed). All algorithms are deterministic under this communication model. The synchronous communication model, although simple, yields interesting insights into the performance of each of the recovery algorithms. Evaluation of our algorithms using a more general asynchronous communication model is currently under investigation. However, we believe an asynchronous implementation will demonstrate similar trends.

We simulate the following scenario:

1) Before t', $\forall v \in V$ $\overrightarrow{min_v}$ and $dmatrix_v$ are correctly computed.
2) At time t', $\overline{v}$ is compromised and advertises a $\overrightarrow{bad}$ (a vector with a cost of 1 to *every* node in the network) to its neighboring nodes.
3) $\overrightarrow{bad}$ spreads for a specified number of hops (this varies by experiment). Variable k refers to the number of hops that $\overrightarrow{bad}$ has spread.
4) At time t, some node $v \in V$ notifies all $v \in adj(\overline{v})$ that $\overline{v}$ was compromised. [7]

The message and time overhead are measured in step (4) above. The pre-computation common to all three recovery algorithms, described in Section III-A, is not counted towards message and time overhead.

A. Fixed Link Weight Experiments

In the next three experiments, we evaluate our recovery algorithms over different topology types in the case of fixed link costs.

1) Experiment 1 - Erdös-Rényi Graphs with Fixed Unit Link Weights: We start with a simplified setting and consider Erdös-Rényi graphs with parameters n and p. n is the number of graph nodes and p is the probability that link (i, j) exists where $i, j \in V$. The link weight of each edge in the graph is set to 50. We iterate over different values of k. For each k, we generate an Erdös-Rényi graph, $G = (V, E)$, with parameters n and p. Then we select a $\overline{v} \in V$ uniformly at random and simulate the scenario described above, using $\overline{v}$ as the compromised node. In total we sample 20 unique nodes for each G. We set $n = 100$, $p = \{0.05, 0.15, 0.25, 0.50\}$, and let $k = \{1, 2, ...10\}$. Each data point is an average over 600 runs (20 runs over 30 topologies). We then plot the 90% confidence interval.

For each of our recovery algorithms, Figure 2 shows the message overhead for different values of k. We conclude that `cpr` outperforms `purge` and 2nd `best` across all topologies. `cpr` performs well because $\overrightarrow{bad}$ is removed using a single diffusing computation, while the other algorithms remove $\overrightarrow{bad}$ state through distance vector's the iterative process of the distance vector algorithm. `cpr`'s global state after rolling back is almost the same as the final recovered state.

2nd `best` recovery can be understood as follows. By Corollary 4 and 5 in Section IV-B, distance values increase from their initial value until they reach their final (correct) value. Any intermediate, non-final, distance value uses $\overrightarrow{bad}$ or $\overrightarrow{old}$. Because $\overrightarrow{bad}$ and $\overrightarrow{old}$ no longer exist during recovery, these intermediate values must correspond to routing loops. Table II shows that there are few pairwise routing loops during 2nd `best` recovery in the network scenarios generated in Experiment 1, indicating that 2nd `best` distance values quickly count up to their final value. [8] Although no pairwise routing loops exist during `purge` recovery, `purge` incurs overhead in its purge phase. Roughly, 50% of `purge`'s messages come from the `purge` phase. For these reasons, `purge` has higher message overhead than 2nd `best`.

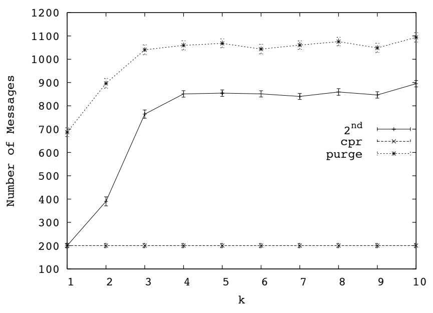
Figure 3 shows the time overhead for the same p values. The trends for time overhead match the trends we observe for message overhead. [9]

[6]Corollary 5 does not apply to `purge` recovery because the $\delta_i(n,d) < \delta_f(n,d)$ condition is not always satisfied.

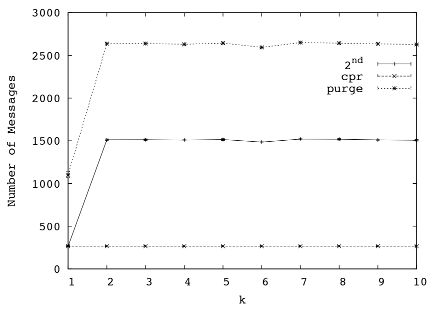
[7]For `cpr` this node also indicates the time, t', $\overline{v}$ was compromised.

[8]We compute this metric as follows. After each simulation timestep, we count all pairwise routings loops over all source-destination pairs and then sum all of these values.

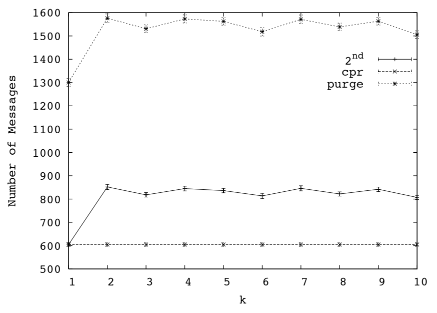
[9]For the remaining experiments, we omit time overhead plots because time overhead follows the same trends as message overhead.



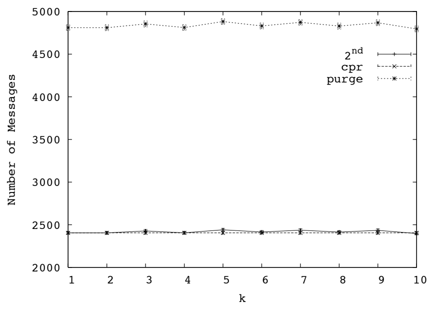
(a) $p = 0.05$, diameter=6.14



(b) $p = 0.15$, diameter=3.01



(c) $p = 0.25$, diameter=2.99



(d) $p = 0.50$, diameter=2

Fig. 2. Experiment 1: message overhead for Erdös-Rényi Graphs with Fixed Unit Link Weights generated over different p values. Note the y-axis have different scales.

	$k = 1$	$k = 2$	$k = 3$	$k = 4 - 10$
$p = 0.05$	0	14	87	92
$p = 0.15$	0	7	8	9
$p = 0.25$	0	0	0	0
$p = 0.50$	0	0	0	0

TABLE II
AVERAGE NUMBER PAIRWISE ROUTING LOOPS FOR 2^{nd} best IN EXPERIMENT 1.

	$k = 1$	$k = 2$	$k = 3$	$k = 4 - 10$
$p = 0.05$	554	1303	9239	12641
$p = 0.15$	319	698	5514	7935
$p = 0.25$	280	446	3510	5440
$p = 0.50$	114	234	2063	2892

TABLE III
AVERAGE NUMBER PAIRWISE ROUTING LOOPS FOR 2^{nd} best IN EXPERIMENT 2.

purge and 2^{nd} best message overhead increases with larger k. Larger k implies that false state has propagated further in the network, implying more paths to repair, and therefore increased messaging. For values of k greater than a graph's diameter, the message overhead remains constant, as expected.

2) Experiment 2 - Erdös-Rényi Graphs with Fixed but Randomly Chosen Link Weights: The experimental setup is identical to Experiment 1 with one exception: link weights are selected uniformly at random between $[1, n]$ (rather than using fixed link weight of 50).

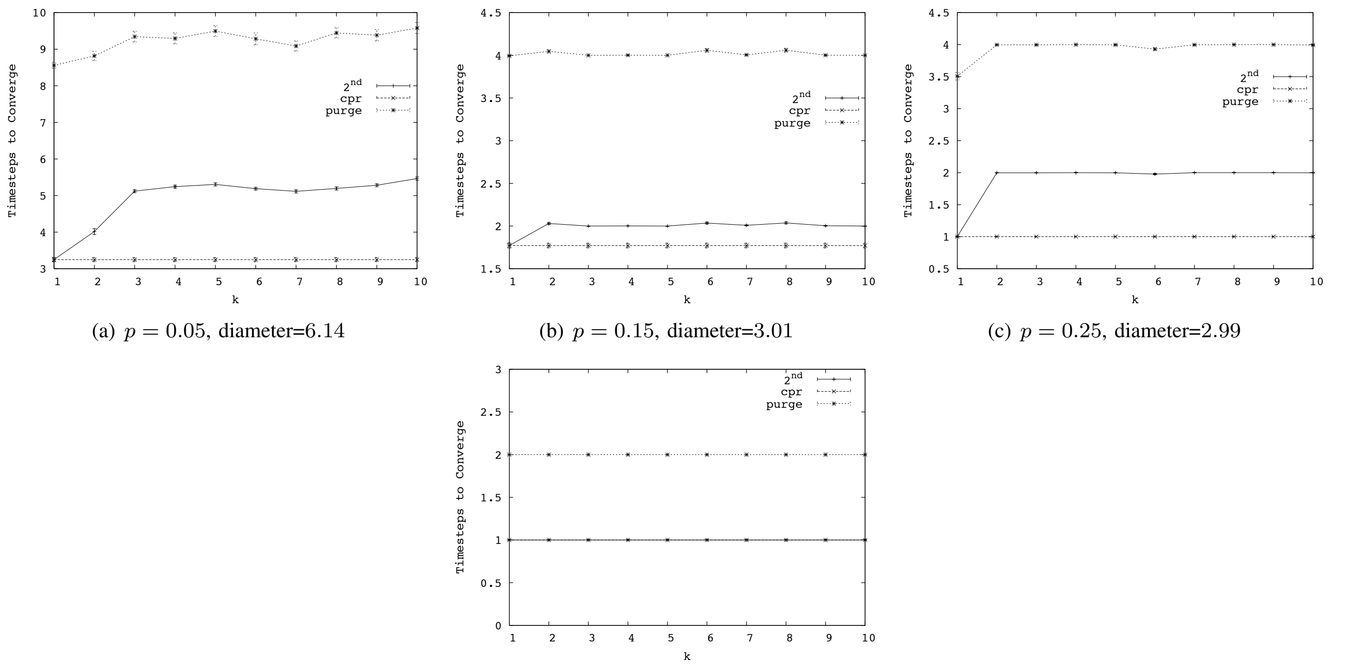
Figure 4 show the message overhead for different k where $p = \{0.05, 0.15, 0.25, 0.50\}$. In striking contrast to Experiment 1, purge outperforms 2^{nd} best for most values of k. 2^{nd} best performs poorly because the count-to-∞ problem: Table III shows the large average number of pairwise routing loops in this experiment, an indicator of the occurrence of count-to-∞ problem. In the few cases (e.g., $k = 1$ for $p = 0.15$, $p = 0.25$ and $p = 0.50$) that 2^{nd} best performs better than purge, 2^{nd} best has few routing loops.

No routing loops are found with purge. cpr performs well for the same reasons described in Section V-A1.

In addition, we counted the number of epochs in which at least one pairwise routing loop existed. For 2^{nd} best (across all topologies), on average, all but the last three timesteps had at least one routing loop. This suggests that the count-to-∞ problem dominates the cost for 2^{nd} best.

3) Experiment 3 - Internet-like Topologies: Thus far, we studied the performance of our recovery algorithms over Erdös-Rényi graphs, which have provided us with useful intuition about the performance of each algorithm. In this experiment, we simulate our algorithms over Internet-like topologies downloaded from the Rocketfuel website [3] and generated using GT-ITM [2]. The Rocketfuel topologies have inferred edge weights. For each Rocketfuel topology, we let each node be the compromised node and average over all of these cases for each value of k. For GT-ITM, we used the parameters specified in Heckmann et al [11] for the 154-node AT&T topology described in Section 4 of [11]. For the GT-ITM topologies, we use the same criteria specified in Experiment 1 to generate each data point.

The results, shown in Figure 5, follow the same pattern as in



(a) $p = 0.05$, diameter=6.14

(b) $p = 0.15$, diameter=3.01

(c) $p = 0.25$, diameter=2.99

(d) $p = 0.50$, diameter=2

Fig. 3. Experiment 1: time overhead for Erdös-Rényi Graphs with Fixed Unit Link Weights generated over different p values.

Experiment 2. In the cases where 2nd `best` performs poorly, the count-to-∞ problem dominates the cost, as evidenced by the number of pairwise routing loops. In the few cases that 2nd `best` performs better than `purge`, there are few pairwise routing loops.

B. Link Weight Change Experiments

So far, we have evaluated our algorithms over different topologies with fixed link costs. We found that `cpr` outperforms the other algorithms because `cpr` removes false routing state with a single diffusing computation, rather than using an iterative distance vector process as in 2nd `best` and `purge`. In the next two experiments we evaluate our algorithms over graphs with changing link costs. We introduce link cost changes between the time $\overline{v}$ is compromised and when $\overline{v}$ is discovered (e.g. during $[t', t]$). In particular, let there be λ link cost changes per timestep, where λ is deterministic. To create a link cost change event, we choose a link (except for all $(v, \overline{v})$ links) whose link will change equiprobably among all links. The new link cost is selected uniformly at random from $[1, n]$.

1) Experiment 4: Except for λ, our experimental setup is identical to the one in Experiment 2. We let $\lambda = \{1, 4, 8\}$. In order to isolate the effects of link costs changes, we assume that `cpr` checkpoints at each timestep.

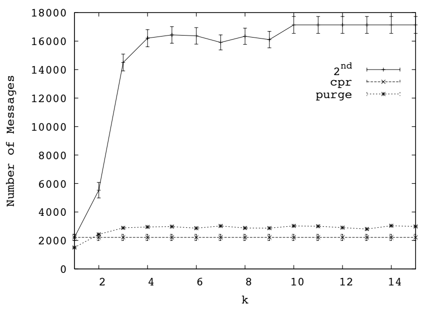
Figure 6 shows `purge` yields the lowest message overhead for $p = .05$, but only slightly lower than `cpr`. `cpr`'s message overhead increases with larger k because there are more link cost change events to process. After `cpr` rolls back, it must process all link cost changes that occurred in $[t', t]$. In contrast, 2nd `best` and `purge` process some of the link cost change events during the interval $[t', t]$ as part of normal distance vector execution. In our experimental setup, these messages are not counted because they do not occur in Step 4 (i.e., as part of the recovery process) of our simulation scenario described in Section V.

Our analysis further indicates that 2nd `best` performance suffers because of the count-to-∞ problem. The gap between 2nd `best` and the other algorithms shrinks as λ increases because as λ increases, link cost changes have a larger effect on message overhead.

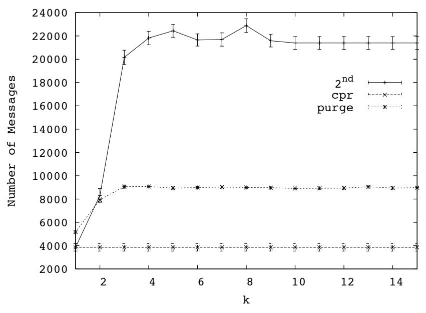
With larger p values, λ has a smaller effect on message complexity because more alternate paths are available. Thus when $p = 0.15$ and $\lambda = 1$, most of `purge`'s recovery effort is towards removing $\overrightarrow{bad}$ state, rather than processing link cost changes. Because `cpr` removes $\overrightarrow{bad}$ using a single diffusing computation and there are few link cost changes, `cpr` has lower message overhead than `purge` in this case. As λ increases, `cpr` has higher message overhead than `purge`: there are more link cost changes to process and `cpr` must process all such link cost changes, while `purge` processes some link cost changes during the interval $[t', t]$ as part of normal distance vector execution.

2) Experiment 5: In this experiment we study the trade-off between message overhead and storage overhead for `cpr`. To this end, we vary the frequency at which `cpr` checkpoints and fix the interval $[t', t]$. Otherwise, our experimental setup is the same as Experiment 4.

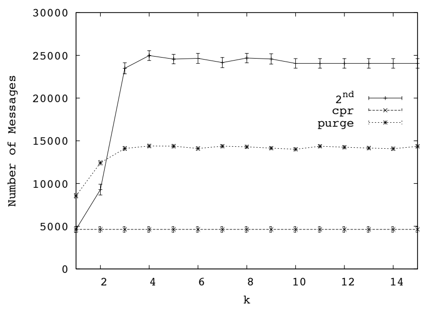
Figure 7 shows the results for an Erdös-Rényi graph with link weights selected uniformly at random between $[1, n]$, $n = 100$, $p = .05$, $\lambda = \{1, 4, 8\}$ and $k = 2$. We plot message overhead against the number of timesteps `cpr` must



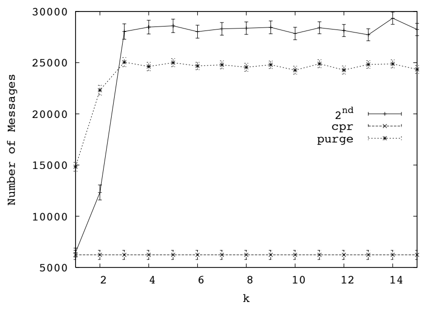
(a) $p = 0.05$, diameter=6.14



(b) $p = 0.15$, diameter=3.01

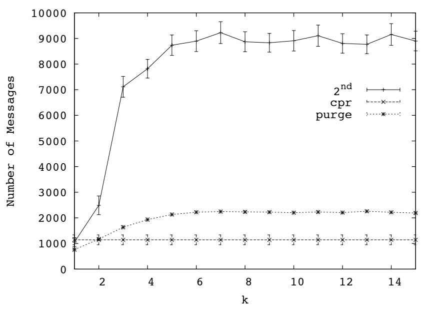


(c) $p = 0.25$, diameter=2.99

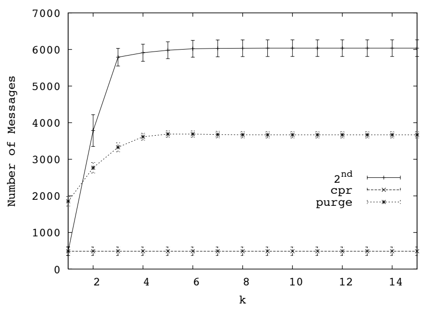


(d) $p = 0.50$, diameter=2

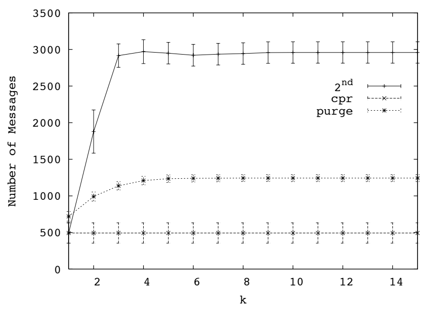
Fig. 4. Experiment 2: message overhead for Erdös-Rényi graph with link weights selected uniformly random from $[1, 100]$. Note the y-axis have different scales.



(a) GT-ITM, $n = 156$, diameter=14.133



(b) Rocketfuel 6461, $n = 141$, diameter=8



(c) Rocketfuel 3867, $n = 79$, diameter=10

Fig. 5. Experiment 3: Internet-like graph message overhead

rollback, z. cpr's message overhead increases with larger z because as z increases there are more link cost change events to process. 2^{nd} best and purge have constant message overhead because they operate independent of z.

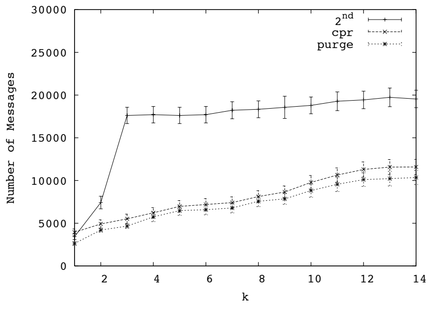
We conclude that as the frequency of cpr snapshots decreases, cpr incurs higher message overhead. Therefore, when choosing the frequency of checkpoints, the trade-off between storage and message overhead must be carefully considered.

C. Summary

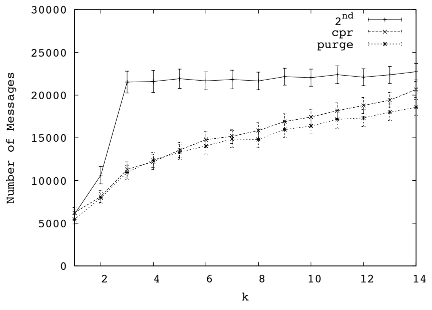
Our results show that for graphs with fixed link costs, cpr yields the lowest message and time overhead. cpr benefits from removing false state with a single diffusing computation. However, cpr has storage overhead, requires loosely synchronized clocks, and requires the time $\overline{v}$ was compromised be identified.

2^{nd} best's performance is determined by the count-to-∞ problem. In this case of Erdös-Rényi graphs with fixed unit link weights, the count-to-∞ problem was minimal, helping 2^{nd} best perform better than purge. purge avoids the count-to-∞ problem by first globally invalidating false state. Therefore in cases where the count-to-∞ problem is significant, purge outperforms 2^{nd} best.

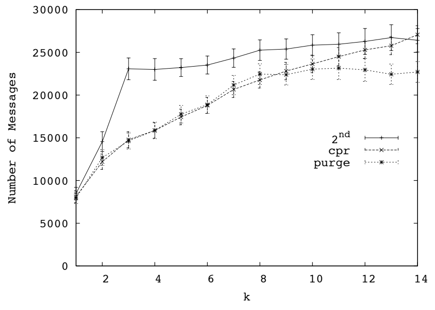
When considering graphs with changing link costs, cpr's performance suffers because it must process all valid link cost changes that occurred since $\overline{v}$ was compromised. Meanwhile, 2^{nd} best and purge make use of computations that followed the injection of false state, that do not depend on false routing state. However, 2^{nd} best's performance degrades because of the count-to-∞ problem. purge eliminates the count-to-∞ problem and therefore yields the best performance over topologies with changing link costs.



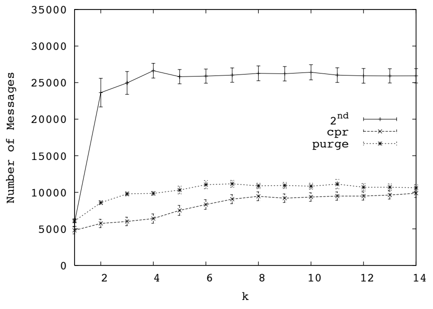
(a) $p = 0.05$, diameter=6.14, $\lambda = 1$



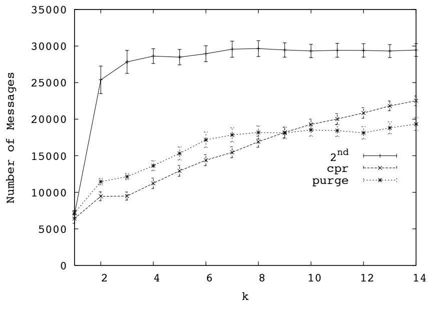
(b) $p = 0.05$, diameter=6.14, $\lambda = 4$



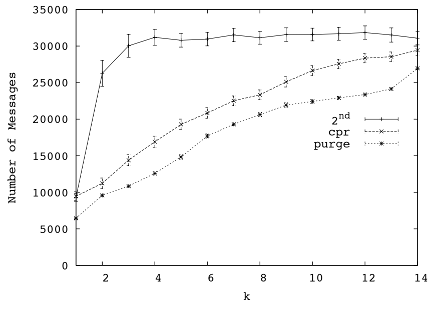
(c) $p = 0.05$, diameter=6.14, $\lambda = 8$



(d) $p = 0.15$, diameter=3.01, $\lambda = 1$

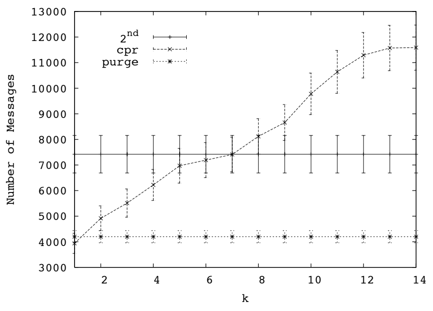


(e) $p = 0.15$, diameter=3.01, $\lambda = 4$

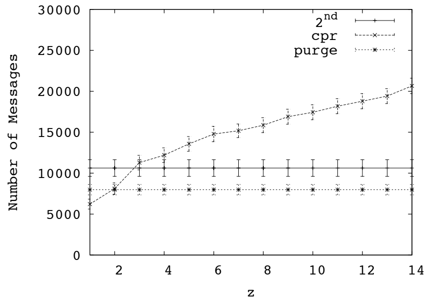


(f) $p = 0.15$, diameter=3.01, $\lambda = 8$

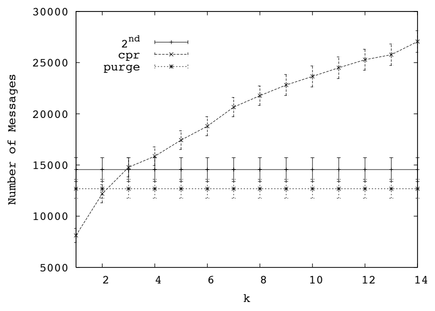
Fig. 6. Experiment 4: Message overhead for $p = \{0.05, 0.15\}$ Erdös-Rényi with link weights selected uniformly random with different λ values.



(a) $p = 0.05$, $k = 2, \lambda = 1$



(b) $p = 0.05$, $k = 2, \lambda = 4$



(c) $p = 0.05$, $k = 2, \lambda = 8$

Fig. 7. Experiment 5: message overhead for $p = 0.05$ Erdös-Rényi with link weights selected uniformly random with different λ values. z refers to the number of timesteps `cpr` must rollback. Note the y-axis have different scales.

Finally, we found that an additional challenge with `cpr` is setting the parameter which determines the checkpoint frequency. More frequent checkpointing yields lower message and time overhead at the cost of more storage overhead. Ultimately, application-specific factors must be considered when setting this parameter.

VI. Related Work

There is a rich body of research in securing routing protocols [12], [20], [23]. However, preventative measures sometimes fail, requiring automated techniques (like ours) to provide recovery.

Previous approaches to recovery from router faults [18], [22] focus on allowing a router to continue forwarding packets while new routes are computed. We focus on a different problem - recomputing new paths following the detection of a malicious node that may have injected false routing state into the network.

Our problem is similar to that of recovering from malicious but committed database transactions. Liu [4] and Ammann [15] develop algorithms to restore a database to a valid state after a malicious transaction has been identified. `purge`'s algorithm to globally invalidate false state can be interpreted as a distributed implementation of the dependency graph approach in [15].

Database crash recovery [17] and message passing systems [7] both use snapshots to restore the system in the event of a failure. In both problem domains, the snapshot algorithms are careful to ensure snapshots are globally consistent. In our setting, consistent global snapshots are not required for `cpr`, since distance vector routing only requires that all initial distance estimates be nonnegative.

Garcia-Lunes-Aceves's DUAL algorithm [10] uses diffusing computations to coordinate least cost updates in order to prevent routing loops. In our case, `cpr` and the preprocessing procedure (Section III-A) use diffusing computations for purposes other than updating least costs (e.g., rollback to a checkpoint in the case of `cpr` and remove $\overline{v}$ as a destination during preprocessing). Like DUAL, the purpose of `purge`'s diffusing computations is to prevent routing loops. However, `purge`'s diffusing computations do not verify that new least costs preserve loop free routing (as with DUAL) but instead globally invalidate false routing state.

Jefferson [13] proposes a solution to synchronize distributed systems called Time Warp. Time Warp is a form of optimistic concurrency control and, as such, occasionally requires rolling back to a checkpoint. Time Warp does so by "unsending" each message sent after the time the checkpoint was taken. With our `cpr` algorithm, a node does not need to explicitly "unsend" messages after rolling back. Instead, each node sends its $\overrightarrow{min}$ taken at the time of the snapshot, which implicitly undoes the effects of any messages sent after the snapshot timestamp.

VII. Conclusions and Future Work

In this paper, we develop methods for recovery in scenarios where a malicious node injects false state into a distributed system. We study an instance of this problem in distance vector routing. We present and evaluate three new algorithms for recovery in such scenarios. Among our three algorithms, our results show that `cpr` – a checkpoint-rollback based algorithm – yields the lowest message and time overhead over topologies with fixed link costs. However, `cpr` has storage overhead and requires loosely synchronized clocks. In the case of topologies with changing link costs, `purge` performs best by avoiding the problems that plague `cpr` and 2nd `best`. Unlike `cpr`, `purge` has no stale state to update because `purge` does not rollback in time. The count-to-∞ problem results in high message overhead for 2nd `best`, while `purge` eliminates the count-to-∞ problem by globally purging false state before finding new least cost paths.

As future work, we are interested in finding the worst possible false state a compromised node can inject. Some options include the minimum distance to all nodes (e.g., our choice for false state used in this paper), state that maximizes the effect of the count-to-∞ problem, and false state that contaminates a bottleneck link. We also would like to evaluate the effects of multiple compromised nodes on our recovery algorithms.

VIII. Acknowledgments

The authors greatly appreciate discussions with Dr. Brian DeCleene of BAE Systems, who initially suggested this problem area.

References

[1] Google Embarrassed and Apologetic After Crash. http://www.computerweekly.com/Articles/2009/05/15/236060/google-embarrassed-and-apologetic-after-crash.htm.

[2] GT-ITM. http://www.cc.gatech.edu/projects/gtitm/.

[3] Rocketfuel. http://www.cs.washington.edu/research/networking/rocketfuel/maps/weights/weights-dist.tar.gz.

[4] P. Ammann, S. Jajodia, and Peng Liu. Recovery from Malicious Transactions. *IEEE Trans. on Knowl. and Data Eng.*, 14(5):1167–1185, 2002.

[5] D. Bertsekas and R. Gallager. *Data Networks*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1987.

[6] E. Dijkstra and C. Scholten. Termination Detection for Diffusing Computations. *Information Processing Letters*, (11), 1980.

[7] K. El-Arini and K. Killourhy. Bayesian Detection of Router Configuration Anomalies. In *MineNet '05: Proceedings of the 2005 ACM SIGCOMM workshop on Mining network data*, pages 221–222, New York, NY, USA, 2005. ACM.

[8] N. Feamster and H. Balakrishnan. Detecting BGP Configuration Faults with Static Analysis. In *2nd Symp. on Networked Systems Design and Implementation (NSDI)*, Boston, MA, May 2005.

[9] A. Feldmann and J. Rexford. IP Network C0onfiguration for Intradomain Traffic Engineering. *IEEE Network Magazine*, 15:46–57, 2001.

[10] J. J. Garcia-Lunes-Aceves. Loop-free Routing using Diffusing Computations. *IEEE/ACM Trans. Netw.*, 1(1):130–141, 1993.

[11] O. Heckmann, M. Piringer, J. Schmitt, and R. Steinmetz. On Realistic Network Topologies for Simulation. In *MoMeTools '03: Proceedings of the ACM SIGCOMM workshop on Models, methods and tools for reproducible network research*, pages 28–32, New York, NY, USA, 2003. ACM.

[12] YC Hu, D.B. Johnson, and A. Perrig. SEAD: Secure Efficient Distance Vector Routing for Mobile Wireless Ad Hoc Networks. In *Mobile Computing Systems and Applications, 2002. Proceedings Fourth IEEE Workshop on*, pages 3–13, 2002.

[13] D. Jefferson. Virtual Time. *ACM Trans. Program. Lang. Syst.*, 7(3):404–425, 1985.

[14] C. Jensen, L. Mark, and N. Roussopoulos. Incremental Implementation Model for Relational Databases with Transaction Time. *IEEE Trans. on Knowl. and Data Eng.*, 3(4):461–473, 1991.

[15] P. Liu, P. Ammann, and S. Jajodia. Rewriting Histories: Recovering from Malicious Transactions. *Distributed and Parallel Databases*, 8(1):7–40, 2000.

[16] D. Lomet, R. Barga, M. Mokbel, and G. Shegalov. Transaction Time Support Inside a Database Engine. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering*, page 35, Washington, DC, USA, 2006. IEEE Computer Society.

[17] C. Mohan, D. Haderle, B. Lindsay, H. Pirahesh, and P. Schwarz. ARIES: A Transaction Recovery Method Supporting Fine-Granularity Locking and Partial Rollbacks Using Write-Ahead Logging. *ACM Trans. Database Syst.*, 17(1):94–162, 1992.

[18] J. Moy. Hitless OSPF Restart. In *Work in progress, Internet Draft*, 2001.

[19] R. Neumnann. Internet routing black hole. *The Risks Digest: Forum on Risks to the Public in Computers and Related Systems*, 19(12), May 1997.

[20] D. Pei, D. Massey, and L. Zhang. Detection of Invalid Routing Announcements in RIP Protocol. In *Global Telecommunications Conference, 2003. GLOBECOM '03. IEEE*, volume 3, pages 1450–1455 vol.3, Dec. 2003.

[21] K. School and D. Westhoff. Context Aware Detection of Selfish Nodes in DSR based Ad-hoc Networks. In *Proceedings of IEEE GLOBECOM*, pages 178–182, 2002.

[22] A. Shaikh, R. Dube, and A. Varma. Avoiding Instability During Graceful Shutdown of OSPF. Technical report, In Proc. IEEE INFOCOM, 2002.

[23] B. Smith, S. Murthy, and J.J. Garcia-Luna-Aceves. Securing Distance-Vector Routing Protocols. *Network and Distributed System Security, Symposium on*, 0:85, 1997.

IX. Appendix

Notation. Let msg refer to a message sent during `purge`'s diffusing computation (to globally remove false routing state). msg includes:

1) a field, src, which contains the node ID of the sending node
2) a vector, $\overrightarrow{dests}$, of all destinations that include $\overline{v}$ as an intermediary node.

Let Δ refer to the maximum clock skew for `cpr`. All other notation is specified in Table I.

Algorithm 1 2nd `best` run at each $i \in adj(\overline{v})$

```
1: $flag \leftarrow$ FALSE
2: set all path costs to $\overline{v}$ to $\infty$
3: for each destination $d$ do
4:     if $\overline{v}$ is first-hop router in least cost path to $d$ then
5:         $c \leftarrow$ least cost to $d$ using a path which does not use
           $\overline{v}$ as first-hop router
6:         update $\overrightarrow{min_i}$ and $dmatrix_i$ with $c$
7:         $flag \leftarrow$ TRUE
8:     end if
9: end for
10: if $flag =$ TRUE then
11:     send $\overrightarrow{min_i}$ to each $j \in adj(i)$ where $j \neq \overline{v}$
12: end if
```

Algorithm 2 `purge`'s diffusing computation run at each $i \in adj(\overline{v})$

```
1: set all path costs to $\overline{v}$ to $\infty$
2: $S \leftarrow \emptyset$
3: for each destination $d$ do
4:     if $\overline{v}$ is first-hop router in least cost path to $d$ then
5:         $S \leftarrow S \cup \{d\}$
6:     end if
7: end for
8: if $S \neq \emptyset$ then
9:     send $S$ to each $j \in adj(i)$ where $j \neq \overline{v}$
10: end if
```

Algorithm 3 `purge`'s diffusing computation run at each $i \notin adj(\overline{v})$

```
Input: $msg$ containing $src$, $\overrightarrow{dests}$ fields.
1: $S \leftarrow \emptyset$
2: for each $d \in msg.\overrightarrow{dests}$ do
3:     if $msg.src$ is next-hop router in least cost path to $d$
       then
4:         $S \leftarrow S \cup \{d\}$
5:     end if
6: end for
7: if $S \neq \emptyset$ then
8:     send $S$ to spanning tree child
9: else
10:     send $ACK$ to $msg.src$
11: end if
```

Algorithm 4 `cpr` rollback

```
1: if already rolled back then
2:     exit
3: end if
4: $\hat{t} \leftarrow -\infty$
5: for each snapshot, $S$, do
6:     $t'' \leftarrow S.timestamp$
7:     if $t'' < (t' - \Delta)$ and $t'' > \hat{t}$ then
8:         $\hat{t} \leftarrow t''$
9:     end if
10: end for
11: rollback to snapshot taken at $\hat{t}$
12: if not spanning tree leaf node then
13:     send rollback request to spanning tree child
14: else
15:     send $ACK$ to spanning tree parent node
16: end if
```

Algorithm 5 `cpr` "steps after rollback" run at each $i \in adj(\overline{v})$

```
1: $flag \leftarrow$ FALSE
2: for each destination $d$ do
3:     if $\overrightarrow{min_i}[d] = \infty$ then
4:         find least cost to $d$ in $dmatrix_i$ and set in $\overrightarrow{min_i}$
5:         $flag \leftarrow$ TRUE
6:     end if
7: end for
8: if $flag =$ TRUE or adjacent link weight changed during
   $[t', t]$ then
9:     send $\overrightarrow{min_i}$ to each $j \in adj(i)$ where $j \neq \overline{v}$
10: end if
```