TESTING IP ROUTING PROTOCOLS – FROM PROBABILISTIC ALGORITHMS TO A SOFTWARE TOOL

Ruibing Hao, David Lee*, Rakesh K. Sinha, and Dario Vlah

Bell Laboratories, Lucent Technologies

Murray Hill, New Jersey 07974

**Bell Laboratories Research China*

Beijing, China

{rhao,lee,rks1,dv}@research.bell-labs.com

Abstract We present probabilistic algorithms for testing IP routing protocols. Different than commercial testing tools, which select their test cases in an ad-hoc manner, our technique chooses test cases with a guaranteed fault coverage. Our algorithms are applied to testing RIP, OSPF, and BGP routing protocols. We test the routing table correctness, database information consistency with network topology and packet forwarding behaviors. We provide an analysis of test execution time and fault coverage. The algorithms are part of a software tool SOCRATES, developed at Bell laboratories. SOCRATES also creates a testing environment so that the generated test cases can be executed in real time on high speed routers.

Keywords: IP protocol, RIP, OSPF, BGP, router fault detection, fault coverage

1. INTRODUCTION

Internet is a packet-switching network that enables its attached computers to exchange information by sending packets to each other. As packets are sent from source to destination, routing decisions are made by special purpose computers, called IP routers. To route packets correctly and efficiently, routers follow a set of rules, which are implemented as distributed algorithms and are called routing protocols [5]. New sophisticated functions and services, such as service differentiation, QoS routing and multicasting, are introduced on a regular basis. Consequently, the design and implementation of routing protocols become

The original version of this chapter was revised: The copyright line was incorrect. This has been corrected. The Erratum to this chapter is available at DOI: 10.1007/978-0-387-35533-7_26 more complex and less reliable. Testing is indispensable for the correct functioning and performance of the routers.

In this work we study testing of router's network layer IP routing protocols. We present algorithms and a software tool for testing Routing Information Protocol (RIP) [4], Open Shortest Path First (OSPF) protocol [7], and Border Gateway Protocol (BGP) [9]. Unfortunately, there is no precise specifications for these routing protocols. The only design requirements are RFCs (Requests for Comments), which tend to be incomplete. Furthermore, vendors often make their own implementation decisions for their routers, which are being deployed throughout the Internet. Consequently, it is out of question to test whether routing protocols are implemented in conformance to the "specification". On the other hand, there are commonly agreed upon "expected behaviors", such as correct routing table, packet forwarding, and network topology information, which are invariant with implementations. There is a practical need and also it is feasible to test whether a router behaves as expected.

Detecting faults in IP router has been an open challenge for both router vendors and network administrators. Most of the systems and tools for detecting router faults can be classified as either passive monitoring tools or active testing tools.

Passive monitoring tools such as RouteMonitor [6] deploy a collection of monitoring devices in a production network to observe the routing traffic exchanged between routers. These monitors do not inject any route update into the network. Based on the observed routing information, these monitors imitate the operation of a real router, execute the routing protocol and compute a routing table. Faults can be found through the comparison of routing tables and analysis of the statistics for each route in the routing table.

Active testing is a more effective approach for fault detection compared with monitoring. Most testing tools test the router in an isolated environment and check the conformance of router's behavior in respect to the RFCs. A main function of these tools is to generate a set of tests that exercise each requirement in the design/RFC for "typical" network configurations/topologies. Because it is computationally infeasible to consider all possible network configurations, any tool can only consider a small subset of all possible configurations. The existing commercial tools pick this subset in an ad-hoc manner so that even if a router passes all their tests, there is no guarantee of the fault coverage.

Our technique is also active testing. The key strength of our approach is that we employ a probabilistic algorithm to select a subset of network configurations. This has several benefits:

1 We can guarantee a high fault coverage.

2 The existing commercial tools use a *fixed* set of network configurations. So if the fault is outside their chosen set, they have absolutely no chance of uncovering the fault. We, on the other hand, consider any configuration with nonzero probability.

3 There is little (if any) benefit of repeating the tests with an existing commercial tool. In contrast, each independent run of our tool picks a different random set of configuration and increases the chances of uncovering a fault. The same effect can be achieved with commercial tools if the user handcodes a different set of configuration each time, but this becomes quite cumbersome especially if the tests need to be repeated multiple times. After all, one of the main goals of test generation tools is to minimize human intervention.

4 We test the router behavior in dynamic environment when routers and networks are constantly going up and down. Again, the choice of which router/network goes up/down is made probabilistically. It is very difficult to manually encode such a huge traffic pattern.

For the test execution on real routers, a testing environment is created so that a Router Under Test (RUT) "perceives" that it is interacting with a real network of routers.

Section 2 introduces a mathematical model of network topologies. Section 3 presents a general probabilistic algorithm for testing routing protocols. Sections 4, 5 and 6 specialize the general algorithm to RIP, OSPF and BGP testing, respectively. Section 7 describes a software tool SOCRATES for automatic testing of routing protocols and reports experimental results. We omit all the proofs and the interested readers are referred to [3].

2. A MATHEMATICAL MODEL OF IP NETWORK TOPOLOGIES

We consider two different graph models of the IP network topologies.
First Model: If routers are connected by point-to-point link (e.g., T1 lines), we represent the topology as a weighted graph, where nodes model routers and edges model the link between routers. We use this model for BGP testing.
Second Model: For multiaccess networks where each router can interface with more than one network and vice versa (e.g., routers connected by Ethernet), we model the topology as a weighted bipartite graph on router and network nodes. We use this model for OSPF testing. Since

this model is more general than the previous one (if each network node is restricted to have degree exactly two, this model becomes similar to the first model) and we will be stating all our algorithms on this model, we describe it in more detail.

A network of internet connections is modeled by a directed graph $G =< V, W, E >$ where V is a set of router-nodes, W is a set of network-nodes, and E is a set of directed edges [1]. Router-nodes represent routers and network-nodes represent networks that connect the routers. The interface between the routers and networks is represented by edges in E where each edge has one end node in V and the other in W. Specifically, an edge (v, w) from a router-node v in V to a network-node w in W represents a router v interfacing with a network w with a cost $c(v, w) > 0$, and an edge (w, v) from a network-node w in W to a router-node v in V represents a network w interfacing a router v with a cost $c(w, v) = 0$. A pair of edges (v, w) and (w, v) correspond to a link between router v and network w. We can replace each such pair by an undirected edge $[v, w]$ to obtain an undirected graph. This undirected graph contains two sets of nodes V and W, and there are no edges between nodes in V (W). Thus it is a bipartite graph, denoted by G_b. Both directed graph G and the equivalent bipartite graph G_b model IP network topologies, and we shall use them interchangeably.

Suppose that we have $|V| = n$ routers and $|W| = m$ networks interconnected. One important question we address is what is the smallest value of m (as a function of n) needed in order to represent all possible network topologies? What we mean is that given any bipartite graph G_b, we can construct an auxiliary graph $G^* =< V, E^* >$ on router-nodes so that two router nodes are connected by an edge iff they connect to a common network node in G_b. Essentially $G^* =< V, E^* >$ represents the "connectivity-pattern" among routers in G_b. Then we determine the smallest value of m such that for each possible G^* on n nodes, there is a G_b with m network nodes that will correspond to this particular G^*. This is a basic problem for other applications such as network simulation. If a chosen m is too small, we miss some network topologies. On the other hand, if m is too large, it introduces redundancy and wastes resources. We present the result:

Theorem 1 *Any network topology can be represented by a bipartite graph $G_b =< V, W, E_b >$ with n router-nodes V and $\lfloor \frac{n}{2} \rfloor \lceil \frac{n}{2} \rceil$ network-nodes W. Conversely, for any bipartite graph $G_b =< V, W, E_b >$ with n router-nodes V and $\lfloor \frac{n}{2} \rfloor \lceil \frac{n}{2} \rceil$ network-nodes W, there is a network topology, which is represented by G_b.*

Therefore, we have a complete representation of network topologies by bipartite graphs with a matching lower and upper bounds on the numbers of network-nodes needed for the representation. Our probabilistic test generation algorithm and fault coverage analysis are based on this representation and its completeness.

3. A PROBABILISTIC TESTING ALGORITHM

We present a general probabilistic method for testing routing protocols. Starting with an empty network topology graph, we probabilistically insert and delete edges and nodes until the graph becomes complete. After each network topology update, we check the RUT for its network topology database, routing table, and packet forwarding behaviors. We first describe the overall probabilistic testing strategy and then summarize in a generic algorithm with an analysis of its run time. In later sections, we specialize the algorithm to testing RIP, OSPF and BGP.

3.1. TESTING STRATEGY

Ideally, we want to check the RUT for all possible network topologies, however, it is impossible since there are too many $(2^{n\lfloor\frac{n}{2}\rfloor\lceil\frac{n}{2}\rceil})$. Instead, we test on a small portion of network topologies, which are generated probabilistically, and we shall show that our approach guarantees a high fault coverage.

Initially we start with a graph $G_b =< V, W, E_b >$ with only one router-node $V = \{v_0\}$, which is the RUT, and $W = E_b = \emptyset$. Depending on the networks where a router is to be deployed, we set an upper bound n on the number of router-nodes V in the network. The number of network-nodes W is bounded above by $c(n) = \lfloor\frac{n}{2}\rfloor\lceil\frac{n}{2}\rceil$ as in Theorem 1. We repeat the following steps until G_b becomes a complete bipartite graph with n router-nodes in V and $c(n)$ network-nodes in W.

1 Randomly insert or delete an edge. An edge insertion means: a new link is added between a router and a network or a down-link has come back. An edge deletion means: a link between a router and a network is down.

2 Randomly insert or delete a router-node (network-node). To insert a router-node (network-node) means: a new router (network) is added, or a crashed router (network) comes back with all its previous links before crash restored. To delete a router-node (network-node) means: a router (network) crashes with all its links to net-

works (routers) down; we remove them from the graph and save them for later restoration.

3 Probability of edge insertion, node insertion, edge deletion, and node deletion are $0 \leq p_1, p_2, p_3, p_4 \leq 1$ respectively; $p_1 + p_2 + p_3 + p_4 = 1$.

4 Keep track of the set $V_0 \subseteq V$ of neighboring router-nodes of v_0, which are connected by a network-node with v_0. Specifically, a router-node v is in V_0 if and only if there exist a network-node w such that $[v_0, w], [w, v] \in E_b$.

5 For each network topology generated above, check the RUT in node v_0:

   (a) Network topology database and routing table.
   For each neighboring router-node v in V_0:

      i Compute its routing update information because of topology change;
      ii Send the computed information to v_0;
      iii Obtain network topology database and routing table information from RUT v_0;
      iv Compute the expected network topology database and routing table of router-node v_0;
      v Compare information from Items iii and iv. A discrepancy indicates a fault.

   (b) Packet forwarding behavior.
   For each neighboring router-node v_i in V_0:

      i Find all the router-nodes v in V such that the chosen path by the routing protocol under test - usually a shortest path - from v_i to v contains v_0, and determine the first node u in V_0 which is on the path after v_0; we have a chosen path $[v_i, v_0, u, \ldots, v]$.
      ii Construct and send an IP packet P from v_i to each such router-node v;
      iii Router-node u is to receive packet P from RUT v_0; otherwise, there is a fault.

3.2. A TESTING ALGORITHM

The testing strategy described above can be summarized as a generic algorithm that applies to routing protocols, including RIP, OSPF and BGP.

For clarity, we do not include parameters and variables in subroutines except for the RUT v_0. We will also use the notation "send an IP packet to router-node v" for "send an IP packet to *a stub network attached to router-node v*".

```
Algorithm 1
input. $n, 0 \leq p_1, p_2, p_3, p_4 \leq 1$.
output. implementation fault or conformance.
1  repeat
2      construct initial network topology graph $G_b$ with
       $V = \{v_0\}, V_0 = W = E_b = \emptyset$;
3  while ($G_b$ is not complete)
4      $UPDATE(v_0)$;
5      if $ROUTE(v_0)$ =FALSE or $FPACKET(v_0)$ =FALSE;
6          return "faulty";
7  return "conforms"
```

Figure 1 Testing Algorithm

The Algorithm is probabilistic in nature. In the next subsection, we will derive the expected number of iterations of the **while**-loop in line 3 (Proposition 1). Each iteration of the **while**-loop guarantees a small fault-coverage. Repeating the test inside the **while**-loop increases fault-coverage. The **repeat**-loop in line 1 runs for a sufficient number of times for a desired fault coverage. The exact number of repetitions needed will be computed later for specific protocol testing. Line 2 constructs an initial network topology graph with only one router-node under test: v_0. Loop in Line 3 continues until a complete bipartite graph is obtained. Subroutine $UPDATE(v_0)$ in Line 4 gets a new network topology, as described in Item 1-3. This subroutine will be described in Section 3.3 with a run time analysis of the algorithm. Subroutines $ROUTE(v_0)$ and $FPACKET(v_0)$ in Line 5 check the routing table and packet forwarding behavior, respectively, of the RUT, as in Items 4(a) and 4(b). If any faults are detected, we abort the process and report "faulty" in Line 6. Otherwise, we declare conformance in Line 7 with a good confidence in the topologies and router behaviors that we have tested. Subroutine $FPACKET(v_0)$ will be discussed in Section 3.4, and $ROUTE(v_0)$ will be described in Section 4, 5 and 6 for RIP, OSPF and BGP, respectively, since it checks different functions for different protocols.

3.3. SUBROUTINE UPDATE(V_0) AND RUN TIME ANALYSIS

We repeat the loop in Line 3 of Algorithm 1 until we obtain a complete bipartite graph. Each repetition of the loop runs the Subroutine $UPDATE(v_0)$ in Line 4, which generates a network topology for a test on the router.

```
Subroutine UPDATE($v_0$)
parameters: $n, m = c(n), 0 \le p_1, p_2, p_3, p_4 \le 1$ with $p_1 + p_2 + p_3 + p_4 = 1$.
variables: $G_b =< V, W, E_b >, V_0$.
1  switch ($p$)
2      case '$p_1$': if ($|E_b| < |V| \cdot |W|$) /* graph is not complete */
3          insert an edge u.a.r. in $E_b$;
4      case '$p_2$': if ($|V| + |W| < n + m$) /* nodes below upper bounds */
5          insert a node u.a.r. in $V \cup W$;
6      case '$p_3$': if ($|E_b| > 0$) /* edge set not empty */
7          delete an edge u.a.r. from $E_b$;
8      case '$p_4$': if ($|V| + |W| > 1$) /* node set not empty */
9          delete a node u.a.r. from $V \cup W$;
10 compute $V_0$; /* neighboring router-nodes of $v_0$ */
11 compute RoutingUpdate;
   /* obtain protocol specific routing update information*/
12 return RoutingUpdate
```

Figure 2 subroutine UPDATE(v_0)

For a network topology update, one of the four operations on edge or node insertion or deletion is performed with probabilities $0 \le p_1, p_2, p_3, p_4 \le 1$. We can partition the unit interval into four subintervals of length p_i, $i = 1, 2, 3, 4$, and then sample uniformly at random (u.a.r.) in the unit interval and obtain $0 \le p \le 1$. We then "switch" on the value of p in Line 1. The run time is summarized below:

Proposition 1 *The expected number of iterations of the* **while**-*loop in Line 3 of Algorithm 1 is at most:*

$$\begin{array}{ll} (n + c(n) + n \cdot c(n) - 1)^2, & \textit{if } p_1 + p_2 - p_3 - p_4 = 0 \\ \dfrac{n + c(n) + n \cdot c(n) - 1}{p_1 + p_2 - p_3 - p_4}, & \textit{if } p_1 + p_2 - p_3 - p_4 > 0 \end{array}$$

where n is the maximal number of router-nodes, $c(n) = \lfloor \frac{n}{2} \rfloor \lceil \frac{n}{2} \rceil$, and p_1, p_2, p_3, and p_4 are the chosen probability of edge insertion, node insertion, edge deletion, and node deletion, respectively. Furthermore, *any network topology with no more than n router-nodes has a non-zero probability of getting tested by Algorithm 1.*

We can choose the four probability distributions so long as $p_1 + p_2 \geq p_3 + p_4$, which guarantees the completion of the algorithm.

3.4. SUBROUTINE FPACKET(V_0) AND PACKET FORWARDING CHECK

Subroutine $FPACKET(v_0)$ at Line 5 in Algorithm 1 tests if the RUT forwards packets correctly. Each packet switched by the router-node under test v_0 must pass through a router-node in the neighboring set V_0, and we only need to check the packet forwarding behavior of v_0 for the packets sent from V_0. For each router-node v_i in V_0, we first find all the router-nodes v in V such that the chosen path by the routing protocol under test - usually a shortest path - from v_i to v contains v_0. We then determine the router-node u in V_0 which is the first node on the path from v_0 to v. Hence, a packet P sent from v_i to v along the path $[v_i, v_0, u, \ldots, v]$ must be received by u in V_0. When we construct and send an IP packet P from v_i to the destination router-node v, the router-node u must receive the packet P from v_0; otherwise, there is a fault and the subroutine returns FALSE.

```
Subroutine FPACKET($v_0$)
input. variables $G_b =< V, W, E_b >, V_0$.
1  for each router-node $v_i$ in $V_0$, $i = 1, \ldots, r$
2      construct SPT $T_i$ rooted at $v_i$;
3      for each router-node $v$ in subtree of $T_i$ rooted at $v_0$
4          send packet $P$ from $v_i$ to $v_0$ with destination $v$;
5          let router-node $u$ in $V_0$ be ancestor of $v$ in $T_i$;
6          if node $u$ does not receive packet $P$ from $v_0$
7                  return FALSE;
8  return TRUE
```

Figure 3 subroutine for testing packet forwarding behavior

Assume that a routing protocol uses shortest path route. (For BGP, a similar scheme can be devised with its notion of "preferred routes.") For each router-node v_i in V_0, Line 2 constructs a Shortest Path Tree (SPT) rooted at v_i, and the router-node under test v_0 is a child of v_i. A packet from v_i to a destination router-node v passes v_0 if and only if v is a descendant of v_0 in the SPT T_i. Line 3-7 checks v_0 for its forwarding packet P from v_i to v. The packet P must be sent from v_0 to u, a node in V_0 and an ancestor of v. Otherwise, a fault in packet forwarding is reported in Line 6-7.

4. TESTING RIP PROTOCOL

We now study the subroutine *ROUTE* for testing RIP [4], a simple distance vector protocol. It uses the asynchronous version of Bellman-Ford algorithm to construct shortest paths to all router-nodes connected to the network. For RIP, $c(u, v)$ - the cost of interfacing from the router-node v to the network-node w is always equal to one. For each destination node, the routing table contains the distance to and also the next-hop to route packets to that destination. RIP intends to cope with dynamic networks with nodes and links up and down. To make sure that distance vectors get updated efficiently and also to avoid routing loops, most implementations use various heuristics such as *Triggered update* and *Split horizon* [5].

Algorithm 1 tests RIP for the packet forwarding behavior in subroutine *FPACKET* with dynamic networks generated by subroutine *UPDATE*. It uses subroutine $ROUTE(v_0)$ in Figure 4 to check whether RIP of the RUT responds correctly to the changed network topology, i.e., whether it constructs a correct distance vector.

```
Subroutine ROUTE($v_0$)
variables. $V_0$. /* neighboring router-nodes of $v_0$ */
1 for nodes in $V_0$ in random permutation: $v_i$, $i = 1, 2, \ldots, r$
2       construct distance vector $D_i$ of $v_i$;
3       send to $v_0$ distance vector: $D_i^* := split\_horizon(D_i)$;
4       construct updated (expected) distance vector of $v_0$:
        $D_0 := D_0 \oplus D_i^*$;
5       obtain distance vector of $v_0$: $\hat{D}_0$;
        /* via RIP protocol interface with $v_0$ */
6       if ($\hat{D}_0 \neq D_0$)
7           return FALSE;
8 return TRUE;
```

Figure 4 subroutine ROUTE for RIP

Now we analyze the fault coverage of Algorithm 1. The analysis has two parts. We first claim that if the algorithm reports a fault then there indeed is a fault in the implementation. The second claim is that if the implementation contains a fault then our algorithm is going to catch it with a high probability. We present the results:

Proposition 2 *If Algorithm* 1 *reports a fault then there is indeed a fault in the implementation of Bellman-Ford algorithm.*

We first describe a fault model and then discuss the fault coverage. Because Bellman-Ford algorithm proceeds by applying a series of relaxation steps, a reasonable fault model is to assume that a relaxation step is computed incorrectly. This is also known as a *single-fault* model. A single fault involves three router-nodes v_1, v_2, and v_3. The idea is that given a triangle consisting of nodes v_1, v_2, and v_3, the implementation, in trying to decide the shortest path from source v_0 to node v_2, incorrectly picks the length two path $v_1 \to v_3 \to v_2$ over edge $v_1 \to v_2$. Given that the implementation contains a single-fault, there are many possible ways in which the testing algorithm can detect this fault:

Theorem 2 Given $0 < \epsilon < 1$, with $e^2.n^2.\ln\frac{1}{\epsilon}$ repetitions in line 1, Algorithm 1 catches any single-fault in a RIP implementation with probability at least $1 - \epsilon$.

5. TESTING OSPF PROTOCOL

We now study a more complex routing protocol OSPF [7]. OSPF is a link state routing protocol. Neighboring OSPF routers maintain adjacency relationship by exchanging "Hello" packets. Each OSPF router generates Link State Advertisements(LSAs) to describe its own network connections and routes learned from other routing protocols. For broadcast network, a designated router is responsible for maintaining adjacency relationships with all other routers on this network. These LSAs are sent to adjacent OSPF routers via flooding. Each OSPF router keeps a LSA database that describes the current network topology, and exchanges its database information with all the neighbors so that each node has the same view of the network topology. Based on the network topology information in the LSA database, each router-node constructs a routing table using shortest paths algorithms.

Algorithm 1 uses subroutine *ROUTE* in Figure 5 to check whether OSPF of the RUT responds correctly to the changed network topology with a link or node up or down. Specifically, it checks: (1) After receiving a link-state advertisement LSA, RUT v_0 constructs a correct link-state database D_0; and (2) v_0 floods a correct link-state advertisement to each immediate neighbor node v_i in V_0. (3) v_0 construct correct routing table from its LSA database.

Whenever a link is down, the two adjacent nodes detect it, and form an LSA to send to all the neighbors. Whenever a router or network node is down, all its neighbors detect it as all the links to that node is down, and flood this information through an LSA. For simplicity, for each

```
Subroutine $ROUTE(v_0)$
input. $LSA$.
variables. $V_0$. /* neighboring router-nodes of $v_0$ */
if (changed link not adjacent to $v_0$)
    select u.a.r. a node $v_r$ in $V_0$ reachable from changed links;
    send $LSA$ from $v_r$ to $v_0$;
else /* changed links adjacent to $v_0$ */
    $v_0$ informed of new link status;
for each $v_i$, $i \neq r$, in $V_0$ /* check LSA flooding from $v_0$ */
    receive $\hat{LSA}$ from $v_0$; /* via protocol interface */
    if $\hat{LSA} \neq LSA$
        return FALSE;
compute expected LSD for $v_0$: $D_0$;
obtain LSD from $v_0$: $\hat{D}_0$; /* via explicit LSD download */
if ($\hat{D}_0 \neq D_0$) /* check LSD of $v_0$ */
    return FALSE;
calculate the expected routing table $R_0$ from this LSD;
obtain routing table $\hat{R}_0$ from $v_0$;
if ($\hat{R}_0 \neq R_0$) /* check routing table of $v_0$ */
    return FALSE;
return TRUE;
```

Figure 5 subroutine ROUTE for OSPF

node or link up or down, we denote this network update information by LSA. We modify Line 4 of Algorithm 1 and collect this information via $LSA = UPDATE(v_0)$.

6. TESTING BGP

The routers within the Internet have been grouped into administrative units called autonomous systems (AS). RIP and OSPF are examples of routing protocols used *within* an AS. BGP [9] is the routing protocol of choice between ASs. Each router maintains its preferred path (called AS-Path) to all possible destinations. Each BGP router advertises these paths to all its adjacent (peer) routers. A key aspect of BGP is that the path used for routing is not necessarily the shortest path. This is done by specifying a set of policies. Each AS can independently set preferences for its neighboring routers. When this AS receives two different routes for the same destination, it picks the route advertised by the router with the higher preference.

A simple BGP testing algorithm can use a subroutine for routing information checking similar to ROUTE(v_0) for RIP protocol testing, and we omit an explicit description of the subroutine. We comment that the only notable differences is that instead of exchanging distance vectors, we now exchange AS-PATHs to each possible destination. We set the routing policy of the RUT and then we check whether it correctly computes its set of preferred paths to all destinations.

Note that the strategy that we have just described only tests the behavior for the policy that we have chosen. Which policy should we pick? A fault may show up only under certain routing policies. We would like to change the policies while the test is in progress with dynamic network topologies; it could give us a larger fault-coverage with respect to policies. However, changing the routing policies of the RUT is feasible only with the software tool for the routers made by some specific vendors.

Policies, in general, are a complex and ill-understood part of the BGP protocol. For example, each AS can set its policy independently. It was shown [10] that policies which appear reasonable locally may be mutually inconsistent. Furthermore, testing whether a given set of routing policies are mutually consistent is NP-hard [2].

There are other difficulties with BGP testing. A design philosophy of BGP is to maintain route stability, so if the network configuration changes very fast, many vendors design their routers to deliberately ignore the route changes. So we are forced to slow down our network simulator to enable the router to "catch up." (In general, this is not a problem with I-BGP testing.)

7. EXPERIMENTAL RESULTS

The probabilistic algorithm for IP router testing has been implemented and is a part of SOCRATES, a software tool for automatic IP routing protocol testing, developed at Lucent Bell Laboratories. The tool is written in ANSI C and runs under Linux operating system. It is inexpensive and completely portable. The only hardware support we require is a workstation capable of connecting to the RUT via at least two Ethernet. Currently SOCRATES supports the testing of RIP (versions 1,2), OSPF (version 2) and BGP (version 4) protocols.

The architecture of SOCRATES is shown in Figure 6. It consists of four components and several auxiliary utilities. The four components are *network topology generator, test executor, test and traffic log* and *GUI.*

Network topology generator implements the probabilistic test generation algorithm. It uses a graph model to represent the network topology, simulates the network links (routers) up or down by adding or removing edges (nodes). For each network topology change, a test case is generated and sent to the *test executor. Test executor* implements the FPACKET

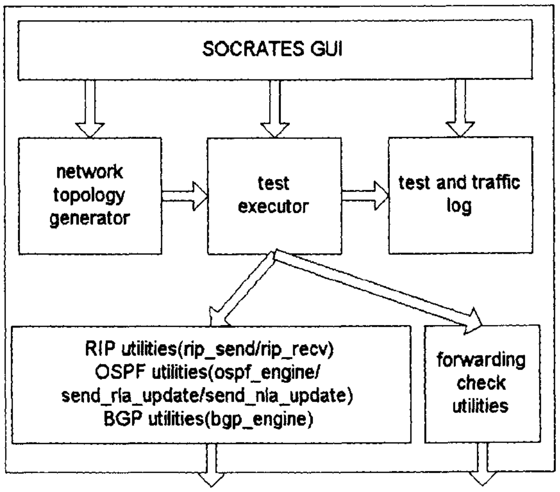
SOCRATES GUI
network topology generator
test executor
test and traffic log
RIP utilities(rip_send/rip_recv)
OSPF utilities(ospf_engine/
send_rla_update/send_nla_update)
BGP utilities(bgp_engine)
forwarding check utilities

Figure 6 Architecture of SOCRATES system

subroutine and the $ROUTE(v_0)$ subroutine for RIP, OSPF and BGP. It executes the test cases and verifies if RUT updates its routing table correctly and forwards IP packets correctly. *Test executor* uses the utilities discussed below to send topology updates to the RUT, queries the RUT to get the updated routing table, performs consistency check with the calculated expected routing table, performs packet forwarding checking if there is a change in the routing table, and makes final test verdict. *Test and traffic log* records all the generated test cases in a log file. For debugging purpose, we also log the IP level trace of all message exchanges between RUT and SOCRATES.

SOCRATES uses some auxiliary utilities to communicate with the RUT and carry out real testing. Except for BGP utility, which is based on TCP connection for packet exchanging, all other utilities use the networking facilities of the host operating system to read all packets on the network in promiscuous mode, and to generate packets from arbitrary source addresses. This allows the emulation of multiple routers, with different IP addresses, from a single test host.

RIP utilities include *rip_send* and *rip_recv*. Whenever there is a change in the network topology during RIP testing, *rip_send* is used to send a RIP update packet (reflecting the change) to the RUT, and *rip_recv* is used to explicitly query the RUT to get the updated routing table.

OSPF utilities includes *ospf_engine*, *send_rla_update* and *send_nla_update*. An *ospf_engine* is a small OSPF kernel, but can be controlled by the *test executor*. Its function is to maintain the fully adjacent relationship between the RUT and itself. It also synchronizes its local LSA database with RUT and responds to all the flooding to make sure it has the exact image of the LSA database in RUT. Whenever there is a change in the network topology during OSPF testing, a new router-LSA is generated to reflect the change. A new network-LSA may be generated depending on which node is the designated router for the affected network in the graph. These two LSAs are sent to the RUT by using *send_rla_update* and *send_nla_update*, respectively.

bgp_engine is a small program for setting up a TCP connection between the RUT's BGP port and the tester. It is controlled by the *test executor* via SOCRATES control protocol to exchange BGP packets with the RUT. Whenever there is a change in the network topology during BGP testing, *bgp_engine* is directed to send a BGP UPDATE packet to the RUT.

Packet forwarding checking utilities are used to verify the RUT's forwarding behavior whenever there is a change in the RUT's routing table.

We used SOCRATES to test the RIP and OSPF implementation of GATED [8], Lucent PacketStar (release 1.2) and CISCO 7206 Router (IOS 11.3). We ran test sessions on a Linux PC with four network interfaces connected to the RUT.

In order to verify that SOCRATES can detect the implementation errors, we introduced a bug into the RIP source code in GATED [8] by modifying the way it processes any RIP update. Specifically, the buggy implementation ignored the last entry of any distance vector it received. We used SOCRATES to test against this buggy implementation, and found that for a configuration of size 20 routers $\times$ 100 networks, SOCRATES could catch this bug in less than one minute.

The RIP standard document [4] was written to summarize already existing implementations. The vagueness of the document permitted the three RIP implementations to add heuristics, in order to improve the performance of the protocol. Unfortunately, this made testing more difficult. On the other hand, OSPF is a well specified protocol with less room for arbitrary modifications. Testing OSPF implementations resulted in successful test sessions with no interoperability issues. The only minor problem we found is that one vendor's router sends out redundant OSPF DD packets during LSA database synchronization. We also found that compared with GATED, commercial routers do a much strict correctness check before accepting OSPF LSAs into their LSA database.

8. CONCLUSION

Testing is indispensable for the reliability of the routers. We describe a general probabilistic testing algorithm that can guarantee a fault-coverage under fault-models. We apply this general testing algorithm to RIP, OSPF, and BGP testing. We provide a run time and fault coverage analysis. Finally, we describe a software tool that incorporates these algorithms and also creates a testing environment such that generated test cases can be executed in real time on high speed routers.

References

[1] D. E. Comer and D. L. Stevens. *Internetworking with TCP/IP, Vol. II.* Prentice-Hall, 1999.

[2] T. G. Griffin and G. Wilfong. An analysis of BGP convergenece properties. In *ACM SIGCOMM*, pages 277–288, 1999.

[3] R. Hao, D. Lee, R. K. Sinha, and D. Vlah. Testing IP routing protocols - from probabilistic algorithms to a software tool. *Bell Labs Tech Memo*, 2000.

[4] C. Hedrick. RFC 1058. URL = `www.ietf.org/rfc.html`, June 1988.

[5] C. Huitema. *Routing in the Internet.* Prentice-Hall, 1995.

[6] D. Massey and B. Fenner. Fault detection in routing protocols. *Proceeding of International Conference on Network Protocols*, pages 31–40, 1999.

[7] J. Moy. RFC 2178. URL = `www.ietf.org/rfc.html`, July 1997.

[8] Merit Networks. Gated. URL = `www.gated.org`.

[9] Y. Rekhter and T. Li. RFC 1771. URL = `www.ietf.org/rfc.html`, March 1995.

[10] K. Vardhan, R. Govindan, and D. Estrin. Persistent route oscillations in inter-domain routing. Technical Report 96-631, USC/ISI, 1996.