Network Single Point of Failure Analysis via Model Finding[1]

Sanjai Narain
Y.-H. Alice Cheng
Alex Poylisher
Rajesh Talpade
1 Telcordia Drive
Piscataway, NJ 08854
732 699 2000
{narain, yhcheng, sher, rrt} @
research.telcordia.com

ABSTRACT

A network component is said to be a single point of failure if its failure causes a critical *end-to-end* requirement to be falsified. Single points of failure are avoided by means of redundancy and various fault-tolerance protocols. However, planning for their avoidance from an end-to-end standpoint is a major challenge because constraints of different types need to be reconciled at and across multiple protocol layers. Thus, it is hard for a network administrator to answer the question "Given a network design D and a critical requirement R, is there a network component whose failure will cause R to be false, and if so, how?" This paper presents an approach to formalizing this question in Alloy and automatically answering it. It illustrates the approach via a realistic fault-tolerant virtual private network.

Categories and Subject Descriptors

D.3.3 [**Programming Languages**]: Language Constructs and Features – *frameworks.*

F.3.1 [**Logics and Meanings of Programs**]: Specifying and Verifying and Reasoning About Programs – *specification techniques, mechanical verification*

General Terms

Management, Performance, Design, Reliability, Security, Languages, Theory, Verification.

Keywords

Network, configuration, specification, model-finding, SAT, single-point-of-failure.

1. INTRODUCTION

A network component is said to be a single point of failure if its failure causes a critical *end-to-end* requirement to be falsified. An example of such a requirement is connectivity between two endpoints. Single points of failure are avoided via path and node redundancy and various fault-tolerance protocols. Multiple independent paths can be configured between endpoints so that if a component along the main path fails, traffic is rerouted along another. Also, multiple components can be clustered so that to the outside world they appear as a single abstract component. If one member of the cluster fails, others take over to preserve the abstraction. However, in a complex network, identifying and removing single points of failure from an end-to-end standpoint is challenging because constraints of different types need to be reconciled at and across multiple protocol layers. Examples of challenges are:

1. **Routing protocol misconfiguration.** Just because an alternative path exists does not mean it will be discovered. Errors in routing protocol configuration can prevent these protocols from discovering alternative paths.
2. **Security misconfiguration.** Just because an alternative path exists does not mean traffic can flow over it. Different links have different security policies so that a packet may be allowed along one path but blocked on the alternative path.
3. **Redundancy not preserved at lower protocol layer.** If redundant resources at one layer are mapped to the same resource at a lower layer, then that resource becomes a single point of failure.
4. **Derivative component failures.** A component failure can cause another component failure. For example, a router failure causes failure of each of its interfaces, but not necessarily vice-versa. One may need to account for multiple failures even though there is just a single "incident".

Thus, it is hard for a network administrator to answer the question:

[1] This research was supported by DHS/HSARPA contract NBCHC050092 on network vulnerability analysis and DoD/DARPA contract W15P7T-050C0R402 on automated network planning.

> Given a network design D and a critical requirement R, is there a network component whose failure will cause R to be false, and if so, how?

This paper shows how to formalize and automatically answer this question in Alloy. The plan is as follows:

1. Express D as a constraint upon component configuration parameters. Use model-finding to compute component configurations satisfying this constraint. This idea has been explored in depth in [1].
2. Define a new component configuration parameter called the *failure* bit. If this is set, the component is in the failed state, otherwise it is in the working state.
3. Express R as a constraint upon component configurations that also takes into account the failure bit.
4. Define a new constraint Spof[2] specifying that there is exactly one component whose failure bit is set, unless it is set by failure of another component.
5. Try to find a model of D & Spof & ~R. If the model is found, it shows how components can be configured to satisfy D and yet contain a single point of failure. If, however, no model is found, one has verified that if components are configured in any way satisfying D, then even if a failure satisfying Spof occurs, R will still be true.

D represents concrete design decisions such as specific protocols and logical architectures into which to integrate these. R represents high-level correctness conditions. This plan is analogous to that for program verification where a correctness condition has to be proved from a representation of concrete data structures and algorithms in the program.

We now illustrate this plan in the context of a *realistic* fault-tolerant virtual private network as shown in Figure 1. The end-to-end requirement is that a host at site A communicate with a server at site B. Gateway routers at each site exchange traffic with a wide area network through their physical interfaces i0 and i1. For privacy, however, logical links called GRE tunnels are set up. These are set up by mapping their logical endpoints to physical interfaces. For example, GRE Tunnel 0 is mapped to the physical interfaces i0 and i2. Two special hub routers are set up and a GRE tunnel is set up between each spoke router and each hub router. The resulting ring structure forms the basis for fault-tolerance: traffic normally flows along the upper two tunnels but should hub router 0 fail, traffic can be redirected along the lower two. Routing protocols RIP or OSPF are configured on links to make the redirection

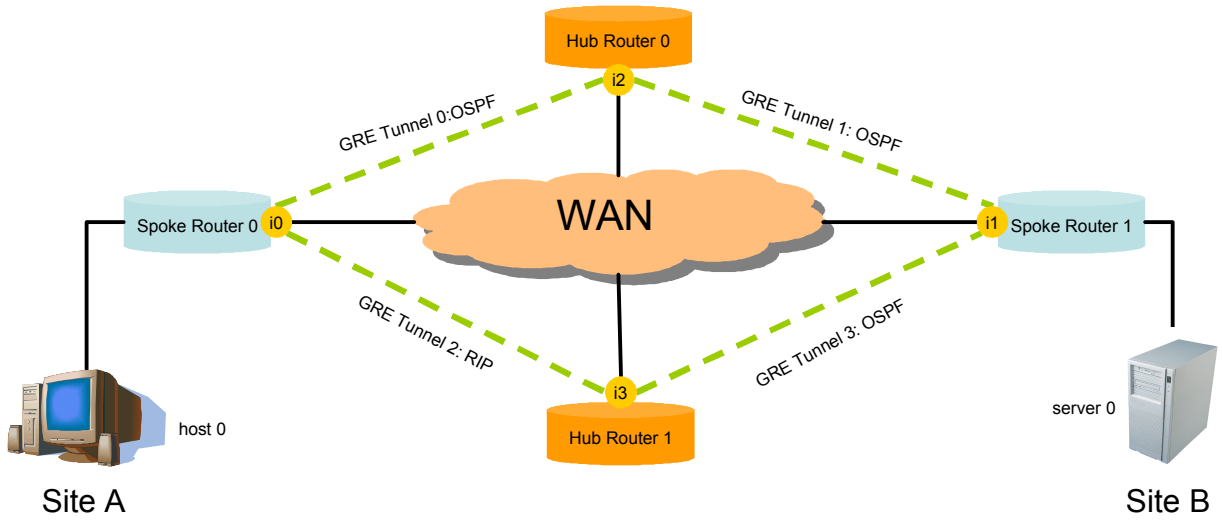

Figure 1. Design of Fault-Tolerant Overlay Network

The introduction of the failure bit after D has been defined needs some clarification. Technically, the failure bit is already part of Alloy signatures modeling components and their configurations. However, D does not constrain this bit at all. Only Spof and R constrain it. This assumption is critical to the correctness of the above plan.

automatic. This design can be captured using the following requirements:

1. There are hub and spoke routers
2. There is a GRE tunnel between every hub and spoke router
3. The host and server are attached to distinct spoke routers
4. A routing protocol is enabled on each GRE link

[2] Spof is abbreviation for single point of failure

This design, while reasonable, suffers from the following shortcomings:

1. Since lower two tunnels have no routing protocols[3] in common, spoke router 0 will not, in fact, discover this alternative route to spoke router 1. Thus, when hub router 0 fails, connectivity between sites A and B would be lost.
2. Since endpoints of GRE Tunnels 0 and 2 at spoke router 0 are both mapped to the same physical interface i0, should i0 fail, tunnels to both hub routers would go down. Thus, connectivity between sites A and B would again be lost.
3. If any of spoke routers should fail then connectivity between sites A and B would be lost.

We now show how to formalize the above design in Alloy, use it to discover above shortcomings, design fixes and finally produce a design overcoming these. Section 2 formalizes the network design, Section 3 the correctness condition, and Section 4 the failure conditions. Finally, Section 5 performs single point of failure analysis, design modification, and verification.

2. FORMALIZING NETWORK DESIGN

The next four declarations define a generic router, a hub and spoke router and a router interface. A router has a failure bit as a configuration parameter. An interface's configuration parameters are the router to which it is attached and its failure bit.

```
sig router {failure: Boolean}
sig hubRouter extends router {}
sig spokeRouter extends router {}
sig interface {chassis: router, failure: Boolean}
```

The next three declarations define a generic routing protocol and two instances of it, OSPF and RIP.

```
sig routingProtocol {}
sig ospf extends routingProtocol {}
sig rip extends routingProtocol {}
```

The following declaration defines a GRE link whose configuration parameters are the two physical interfaces and the routing protocol running over that link.

```
sig greLink {local: interface, remote: interface, routing: routingProtocol}
```

The next two declarations define a host and server whose configuration parameters are the gateway routers through which they communicate with the outside world.

```
sig host {gateway: spokeRouter}
sig server {gateway: spokeRouter}
```

Requirement 1 in the design of the previous section is captured by the declaration of hub and spoke routers, and constraining the scope to contain a non-zero number of routers of each type. The predicate below captures the remainder of design requirements. The first and second clauses capture Requirement 2. The third captures Requirement 3. Requirement 4 is already captured by the declaration of the routing protocol configuration parameter in greLink. Note that Design *does not constrain the failure bit.*

```
pred Design (){
{all r1:hubRouter, r2: spokeRouter |
        some i1, i2:interface, g:greLink | g.local=i1 &&
        g.remote=i2 && i1.chassis=r1 &&
        i2.chassis=r2}
{all x: greLink | x.local !=x.remote}
{all x: host, y:server | x.gateway !=y.gateway}}
```

If we now request Alloy to find a model of Design in the scope consisting of 4 router, 4 greLink, 2 Boolean, 2 routingProtocol, 4 interface, 1 host, 1 server, Alloy returns the following configuration parameters, effectively, implementing the design in Figure 1:

Table 1. A Solution for Constraint Design

Configuration parameter	Component	Value
chassis	interface_0	spokeRouter_0
	interface_1	spokeRouter_1
	interface_2	hubRouter_0
	interface_3	hubRouter_1
local	greLink_0	interface_2
	greLink_1	interface_2
	greLink_2	interface_3,
	greLink_3	interface_3

[3] Routing protocol processes at routers acquire global knowledge about network topology by exchanging local topology knowledge with neighboring routers. They then compute optimum paths from source to destination.

remote	greLink_0	interface_0
	greLink_1	interface_1
	greLink_2	interface_0
	greLink_3	interface_1
routing	greLink_0	ospf_0
	greLink_1	ospf_0
	greLink_2	rip_0
	greLink_3	ospf_0
gateway	host	spokeRouter_0
	server	spokeRouter_1

3. FORMALIZING CORRECTNESS CONDITION

We now define the correctness condition that *does constrain the failure bit.* The predicate below defines what it means for a virtual link between routers x and y to be up, and running a routing protocol p. The meaning is that there is a GRE link between interfaces at these routers, both interfaces are working, and that GRE link is configured with the routing protocol p.

```
pred upLink (x: router, y: router, p: routingProtocol)
{       some l: greLink |
        (l.local.chassis=x && l.remote.chassis=y ||
         l.local.chassis=y && l.remote.chassis=x) &&
         l.local.failure=false &&
         l.remote.failure=false &&
         l.routing=p}
```

The predicate below states that two routers are reachable if at least one path between these consisting of up virtual links can be traced, and each link on that path is configured with the same routing protocol. Alloy's ^ operator computes transitive closure of a relation.

```
pred reachable(x: router, y: router) {
{some p:routingProtocol |
        x in y.^{u, v: router | upLink(u, v, p)}}
```

Since hosts and servers are directly connected to the spoke routers, the correctness condition is simply that all spoke routers are reachable.

```
pred Correctness () {all x,y : spokeRouter |
reachable(x,y) }
```

4. FORMALIZING SINGLE POINT OF FAILURE CONDITIONS

4.1 Single Hub Router Failure

The predicate below defines failure of a single hub router. The second clause states that no spoke router is failed. The third clause states that an interface is failed only if it is on a failed router.

```
pred singleHubRouterFailure ()
{
        (one x:hubRouter | x.failure=true) &&
        (no x: spokeRouter | x.failure=true) &&
        (no x: interface | x.chassis.failure=false &&
x.failure=true)}
```

The fact that a router failure causes failure of each of its interface is expressed via the global constraint:

```
fact {(all x:router | x.failure=true => all u:interface |
u.chassis=x => u.failure=true)}
```

4.2 Single Interface Failure

The predicate below defines failure of just a single interface, and of no router.

```
pred singleInterfaceFailure ()
{       (one x: interface | x.failure=true) &&
        (no x:router | x.failure=true)
}
```

4.3 Single Router Failure

The predicate below defines failure of a just a single router (hub or spoke) and of no interface not on the failed router.

```
pred singleRouterFailure ()
{(one x:router | x.failure=true)
   (no x: interface | x.chassis.failure=false &&
x.failure=true)}
```

5. SINGLE POINT OF FAILURE ANALYSIS

5.1 Single Hub Router Failure

Requesting Alloy to find a model of (Design & SingleHubRouterFailure & !Correctness) for the scope in Section 2, yields the configuration of Table 1, with only

hubRouter_1 and its interface_3 in failed state. These failures disable both upper GRE tunnels originating at hubRouter_1. However, traffic cannot be rerouted along lower tunnels because routing protocols are not identical on these. This problem can be fixed by strengthening Design with a new constraint that configures each GRE link with the same routing protocol: IdenticalRoutingProtocol={some p:routingProtocol | all x: greLink | x.routing = p}.

When (Design & IdenticalRoutingProtocol) & SingleHubRouterFailure & ! Correctness is submitted to Alloy, it cannot find a model. Thus, any network satisfying: Design & IdenticalRoutingProtocol will maintain Correctness across one hub router failure.

5.2 Single Interface Failure

While the previous design is resilient to a hub router failure, it is not so to an interface failure. Requesting Alloy to find a model of (Design & IdenticalRoutingProtcol) & SingleInterfaceFailure & ! Correctness yields a configuration in which interface_0 on spokeRouter_0 is failed. Since both tunnels originating at spokeRouter_0 are mapped to interface_0, both of these are disabled. This problem can be fixed by increasing the number of physical interfaces to 8, 2 per router, and ensuring that no GRE tunnels share a physical interface. This latter constraint is expressed as: DisjointGREInterfaces = {all disj x,y : greLink | x.local != y.local && x.remote !=y.remote}.

When (Design & IdenticalRoutingProtcol & DisjointGREInterfaces) & SingleInterfaceFailure & ! Correctness is submitted to Alloy, it cannot find a model. Thus (Design & IdenticalRoutingProtcol & DisjointGREInterfaces) is a new design that is resilient to two types of single points of failures.

5.3 Single Router Failure

Unfortunately, the previous design is still not resilient to a spoke router failure. When the constraint (Design & IdenticalRoutingProtcol & DisjointGREInterfaces) & SingleRouterFailure & !Correctness is submitted to Alloy, it returns a configuration in which spokeRouter_0 has failed, disabling both interfaces and both originating GRE tunnels. The solution is to create a cluster of more than one router and make them appear as a single virtual router. One method of accomplishing this is via the HSRP protocol. Its inclusion can be specified and verified in the same manner as above.

6. SUMMARY

A network component is said to be a single point of failure if its failure causes a critical *end-to-end* requirement to be falsified. Identifying and removing such points of failure is an inherently hard problem because several different types of constraints at and across multiple protocol layers need to be reconciled. This paper presents an approach for formalizing this problem and automating its solution via Alloy's model finding technology. The approach is illustrated in the context of a realistic fault-tolerant virtual private network.

Strategies to scale this approach were recently investigated in a DARPA project W15P7T-050C0R402. It was shown that the Kodkod model-finder engine [2] could handle scopes of up to 100 routers.

REFERENCES

[1] Network Configuration Management via Model Finding. *Procedings of USENIX Large Installations Systems Administrator Conference (LISA),* San Diego, CA, 2005. Full report available at http://alloy.mit.edu/papers/NetConfigAlloy.pdf#search=%22Narain%20Alloy%20MIT%22

[2] Kodkod model finder for first order relational logic. http://web.mit.edu/~emina/www/kodkod.html