Can you re6st?

re6st is a multiprotocol random mesh generator that uses babel routing protocol to discover optimized routes between each point in the mesh. It supports IPv6 and IPv4 with RINA support coming soon and is used commercially by VIFIB distributed cloud provider to solve the current lack of reliability of Internet for distributed enterprise applications: bugs in routers, packet inspection that breaks TCP protocol, government filtering that filters too much, etc. Without re6st, it would have been impossible to deploy critical business applications used by large companies (Mitsubishi, SANEF, Aide et Action, etc.) on a decentralized cloud. It would also be impossible to manage deployment of distributed cloud in Brazil, China or Ivory Coast where Internet is even less reliable.

The current implementation of re6st has three limitations:

1. it does not scale to more than what linux kernel can support in terms of routing table
2. it does not prevent intruders from sending rogue messages that break the routing protocol
3. it does not prevent deep packet inspection to alter metrics discovered by the routing protocol

This post summarizes a paper addressing the first limitation through a random recursive mesh architecture inspired by the idea of recursivity and unification that we learnt by reading the introduction materials of John Day to RINA. It was financed by the Pristine project, which aims to developing a Software Development Kit for the IRATI Linux prototype.

Understanding re6st

Re6st generates a random mesh of tunnels between computers located all over the world. A certain number (ex. 10) of arrows (tunnels) are randomly created from each node (computer) of the mesh to another node. Each arrow (tunnel) of the mesh is used to carry payloads between nodes (computers) of the mesh. Every given time period (ex. 5 minutes), a certain share of arrows (tunnels) is destroyed and replaced by randomly chosen tunnels.

The best possible combination of arrows to reach another node is computed in real time by a routing protocol, in our case babel. The general design of re6st is modular in abstract: tunnel technology can be changed (ex. openvpn, gre, fou, tinc, etc.) as well as routing protocol (ex. babel, batman, etc.) or protocol (ex. IPv6, IPv4, RINA, etc.).

The general ideas of re6st are described in the report of Ulysse Beaugnon: http://community.slapos.org/P-ViFiB-Resilient.Overlay.Network/Base_download. This paper also demonstrates how re6st can provided better resiliency and reliability than default IPv4 routes on Internet.

Three key principles must be kept in mind:

- randomness creates resiliency
- life requires destruction
- trust no arrow or node

Practical design leads to wrong design

This means that the idea of selecting the « best tunnels » rather than following randomness or the idea of preserving the « best tunnels » rather that taking random destruction decisions will break the resilient nature of re6st and thus make it just useless, since it would then replicate the current non resilient behaviour of Internet. The importance of randomness can be understood by comparing the random decision in re6st to the random choice of stock in an investment portfolio. Selecting the « largest cap » stock (S&P500, CAC40, etc.) will usually provide higher return on investment over a short period of time but higher risk too. Selecting random stock minimizes the risk while providing a fair return on investment, often better than any other heuristic. This is often described by the story in which a monkey selecting random stock ends up with the highest return on investment after a long period of time than any professional trader.

However, there are a few cases where some kind of selection or preservation is required for practical reasons: small sized mesh and tunnels « being used ».

In small sized mesh, randomly selectable arrows are too few. The probability that random selection of arrow selects only poorly performing tunnels becomes too high. For example, a user of VIFIB's operated re6st in China must force re6st to select at least one a arrow which destination is also in China or which does not use openvpn tunnel technology. Else the random selection algorithm will likely select arrows which destination is outside China and which use a tunnel technology (openvpn) that is blocked by the great firewall. For practical reasons, we have in this case to distort the random selection of arrows by favouring arrows which destination is in China or which tunnel technology is not filetered by the great firewall. If we did not do, we would have to wait too long (ex. one hour) until random selection of arrows finds a well performing tunnel. The situation we described in China is similar in other countries even though it is less obvious.

The case of tunnels « being used » is even more critical: if we destroy arrows randomly while they are being used, it takes about one minute for a routing protocol like babel to find a new route. This delay can be reduced by increasing the frequency

of route updates by babel. However, if the number of arrow hops to go from one node to another node in the mesh increases, so does the probability that an arrow is being destroyed during the exchange of packets between two nodes. In a real world situation, users using an accounting software hosted on a re6st mesh would then have to wait about one minute every hour to access a transaction. This is not an acceptable user experience. Therefore, we had to introduce the idea that we try not to destroy arrows being used and to improve the destruction process of arrows. In order to achieve this, we introduced an API to babel in order to collect the routing metrics for every arrow being used. Before destroying an arrow which supports a route present in the routing table, re6st firsts overwrites the metric of the route in babel to simulate a very poorly performing route. Then it waits for this route from being removed from the routing table by babel. Only then it destroys the tunnel. However, nothing proves that the route will be removed, especially in the case of small sized meshes. We therefore set a limit (ex. one hour) after which we forcibly destroy the arrow in order to preserve our principle « destruction creates life ».

Being practical is required to make re6st usable: no user can accept to wait one minute every hour, even for the sake of resiliency. But being practical can not be a valid guideline: key principles of re6st must be preserved else resiliency will be lost. Achieving resiliency is often a matter of being able to handle those few cases which happen during one day every other year. Any reasoning based on the idea of « covering the 99 % case that matter to us » is thus mostly wrong.

Here are some typical wrong ideas:

- group nodes per region to achieve scalabiliy or performance
- group nodes per telecommunication provider to achieve scalabiliy or performance

  group nodes per type of usage (ex. email, http, hdfs, etc.) to achieve scalabiliy or performance

Each of these ideas break the key principle of randomness in the selection of the mesh and thus destroy the resilient nature of re6st.

Here are other typical wrong ideas:

- only destroy arrows that are not being used
- only destroy arrows with bad metrics
- never destroy arrows with good metrics

Each of these ideas also breaks the principle of randomness in the selection and the principle that life requires destruction. Imagine for example that one arrow stays alive because it provides excellent performance. This creates a weakness for a man-in-the-middle intrusion, since the path taken by packets becomes deterministic. Another reason for strict random destruction is that in exceptionnal situations, those which are outside the « 99 % of cases that matter to us », bad arrows in normal times can suddenly become good arrows in crisis times, just like the countryside road becomes the only way to drive whenever all tolls are blocked during a toll strike.

Because customers do not care the reason why a crisis situation happens somewhere and only care about keep on using an online service, being able to handle those crisis situations (which happen about once or twice or year) is one of the reasons to exist of re6st.

Preserving design principles through practical exceptions

In order to maintain the resiliency of re6st, it is essential to maintain its key principles very clear and visible in its code: else any contributor – based on an intuitive sense of praticality – will introduce changes that destroy resiliency and will make the code look like the underlying principles are not those that should be. This will lead to wrong intepretation of re6st, misundertsanding and overall it will turn those exceptions into what appear as principles of re6st.

About the random destruction of arrows:

- Every certain time period (ex. 5 minutes) a certain share (ex. 30%) of arrow is destroyed ;
- As an exception, if an arrow is being used, instead of being destroyed, it is marked as « poorly performing » until the routing protocol eliminates it from the routing table, and is destroyed either whenever it is no longer used or after a longer time period if it is still used (ex. 60 minutes)
- As an exception, a certain share of most used arrows (ex. 20 %) out of the total number of arrows, is destroyed with a lower probability that the other share.

About the random selection of arrows:

- Arrows are selected randomly.
- As an exception, certain arrows can be specified for bootstrap purpose.

- As an exception, a subset of arrows (ex. nodes in well known datacenters) can always be introduced in the random selection process for a certain share of candidates (ex. 30%).

Introducting the n-order mesh

The first ideas that came out to make re6st scalable is to build a hierarchical network on top of a set of re6st meshes. This idea mimics the approach in which OSPF routed autonomous systems (AS) are inteconnected by BGP through a few gateways in each AS.

However, this kind of approach does not satisfy the topology of re6st in the use case of massively distributed cloud: each node may be for example to a home FTTH with limited uplink (ex. 50 Mbps). Turning one node into a gateway would lead to the exhaustion of resources on that node.

Also, we do not like the idea to use different protocols at different levels. This sounds like if we did not learn from the RINA ideas that demonstrate that no rationale justifies having a different API at different levels in a network architecture.

We therefore thought: how would it be possible through a single routing protocol to implement a kind of mesh of mesh with no gateway and no trafic centralization ? The solution to this problem came with the idea of default route propagation in babel. The babel protocol can progate default from one node to another node. The propagation algorithm favours the closest default route in terms of latency for example.

2-order mesh

Here is now the solution for a 2-order mesh. If one node is part of two meshes, it can propagate the address range of one mesh to the other mesh, and vice-versa. As long as we select in every mesh a share (ex. 10 %) of nodes that is also part of another mesh, we can manage through a single routing protocol a quadratic number of nodes with a routing table that has has only double size compared to the routing table for a single mesh. Nodes which belong to two mesh have a routing table that is three times the size of routing table for single mesh. Nodes which belong to three mesh have a routing table that is four times the size of routing table for single mesh.

The choice of which nodes are part of more than one mesh would be similar to a coordinator election in a distributed system, with a calculated value, yet to define how, reflecting the capacity of each node to belong to several meshes. The under-layer protocol could be the same as the one used by re6st to revoke nodes, with some randomness in this choice for better resiliency.

As we can see in this example, there is no dedicated mesh for routing mesh. We implement the mesh of mesh by making nodes belong to two mesh and by using distribution of routing for a default address range. This case was easy because we use for nodes the node address range and for mesh the mesh address range.

3-order mesh

The case of a 3-order mesh requires to group address ranges in higher order address ranges. This implies for now to split one mesh into sub-meshes, each of which with its smaller address range. The choice of address range depends on the protocol that is used as basis for routing.

We then apply the same ideas inside every mesh. In every submesh, a certain share (ex. 10%) of randomly selected nodes also belongs to another submesh and distribute the address range of that submesh. If the total number of nodes in the mesh of mesh of mesh is cubic, then the size of the routing table is linear with a constant that depends on the number of mesh to which a node pertains to.

Visual Example

Here is an example of ungrouped nodes :

n-Order Re6st - Scaling Resiliency | Network Nodes ungrouped

Nodes grouped by colour :

Mesh for red nodes :

n-Order Re6st - Scaling Resiliency | Network Nodes red mesh

Mesh for green nodes :

n-Order Re6st - Scaling Resiliency | Network Nodes green mesh

Mesh for blue nodes :

n-Order Re6st - Scaling Resiliency | Network Nodes blue mesh

All mesh (not interconnected) :

Connecting red mesh and blue mesh to green mesh by selecting blue nodes part of green mesh and red nodes part of green mesh :

n-Order Re6st - Scaling Resiliency | Network Nodes grouped mesh

Blue nodes are never very far from green nodes :

Conclusion

It is possible without breaking re6st core principles for resiliency to achieve scalability thanks to an n-order mesh structure based on the idea of route propagation between randomly selected nodes.