Overlay techniques in the underlay

1st Conference Seminar
Massively Distributed Systems
Winter Term 2006/2007

Elmar Hoffmann
RWTH Aachen University
elho@elho.net

ABSTRACT

Traditional routing algorithms as used today in the Internet rely on a hierarchical structure of the network to be able to scale reasonably. Similarly, routing algorithms in wireless mesh networks that do not rely on a location-imposed hierarchy do not scale well. Routing algorithms as used in overlay networks for Distributed Hash Tables (DHTs), however, have shown good scalability.

This paper describes two novel approaches that take these overlay network routing algorithms and techniques and apply them to the routing in the underlying network. Namely Virtual Ring Routing [3], aimed at providing scalable mesh routing offering good performance, and Routing on Flat Labels [4], aimed at demonstrating that the use of routing schemes on the Internet, that do not rely on a hierarchical structure, could be feasible.

Categories and Subject Descriptors

C.2.2 [**Computer-Communication Networks**]: Network Protocols—*Routing protocols*; C.2.6 [**Computer-Communication Networks**]: Internetworking; C.2.1 [**Computer-Communication Networks**]: Network Architecture and Design

General Terms

Algorithms, Design

Keywords

Routing, naming, Internet architecture, Distributed Hash Tables

1. INTRODUCTION

The scalability of the underlying routing used in a network is an important factor limiting the size of the network. This is a predominant problem in mesh routing protocols used in wireless ad hoc networks [1], where every node also is a router and thus — in practice — scaling issues show up with relatively small numbers of nodes. It also affects the inter-domain routing in the core of the Internet, where routers do not have a default route, but routes to every reachable network in their routing table. This core is therefore also called the default-free zone (DFZ).

It is — besides the available address space — one of the factors that limit the future growth of the current Internet. This is especially of concern due to the rate of advancement in the development of the highly specialised hardware components used in the forwarding engines of high-speed routers. Unlike that of mass produced components, which follows Moore's Law [5], it is considerably lower.

In the early days of the Internet, routing table size showed characteristics of exponential growth. In order to counter this trend, Classless Inter-Domain Routing (CIDR) was developed, which introduced a hierarchy into inter-domain routing. These hierarchy levels, following the underlying hierarchical provider-subscriber topology, allowed for aggregation of routes and thus — at first — limiting the growth rate of the number of routes announced in the DFZ [10, 6, 11].

Since then, hierarchical structure and better aggregation were the main approach to fight the scalability problem of excessive routing table growth. As such, IPv6 address allocation was designed to be strictly hierarchical, provider-assigned (PA) only [9, 8].

However, various factors worked against the CIDR aggregation efforts, again making the routing table size grow at a faster rate. Increasing economic interest in the Internet led to an accelerated growth of it overall as well as to increasing demand for multi-homing and traffic engineering. The latter two result in de-aggregation by announcing more specific routes into the DFZ. Another reason for this to happen is the practice of re-homing without renumbering [10].

Except of the latter, these problems will likewise affect IPv6. The much larger available address space removes the major limiting factor of overall growth of the (IPv4) Internet and allows for use of globally routable addresses in new areas, such as cell phones and other mobile devices. The demand for multi-homing and traffic engineering remains and

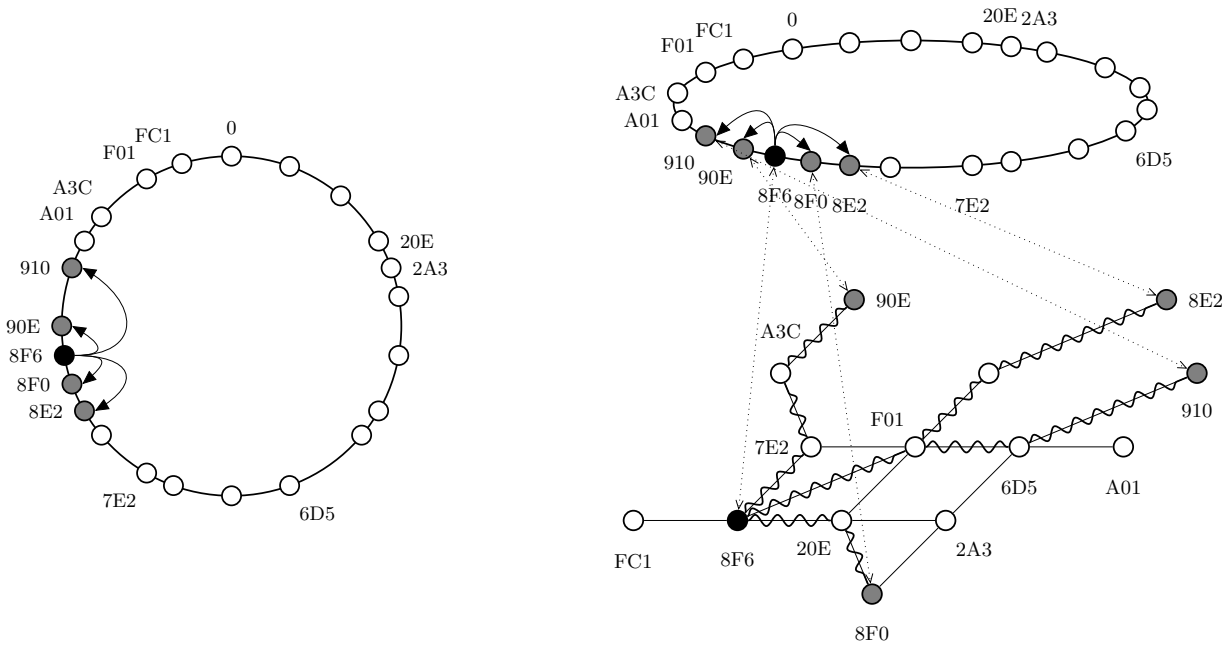

Figure 1: The virtual ring and its relationship to the physical network topology.

thus there has been a shift to allow assignment of provider-independent (PI) address space recently. This of course contradicts the original design of strictly hierarchical address allocation and hampers aggregation, so that the problem of routing scalability remains [5].

In contrast to this, Distributed Hash Tables (DHTs) have demonstrated scalability with a large number of (routing) nodes. Up to now, DHTs were used in overlay networks that rely on an underlying network, also called an underlay network. The two routing algorithms described in the following, however, use DHT algorithms directly on the underlay network.

Another fundamental problem of the current Internet architecture is that the IP address of a node comprises both identity and location information [13]. This is the reason behind the problems faced with multi-homing and re-homing — the location of nodes changes, but their identities should remain the same. Using IP addresses, however, only one of these contraints can be fulfilled. Location independent identifiers solve this conflict and furthermore allow for mobility of nodes without requiring any indirection layers.

This paper will first describe Virtual Ring Routing [3] in Section 2, designed for efficient and scalable mesh routing in ad-hoc networks, and then describe Routing on Flat Labels [4] in Section 3, designed for use in the Internet, among others borrowing ideas from Chord [14], Canon [7] and Virtual Ring Routing.

2. VIRTUAL RING ROUTING

2.1 Overview

Virtual Ring Routing (VRR) uses unique, location independent node identifiers. Each node gets a random unsigned integer as identifier. The size of this identifier can be chosen depending on the application, e. g. it could be 32-bit or 128-bit to provide compatibility with IPv4 or IPv6 addresses respectively or it could be 160-bit and based on a SHA-1 hash value.

Nodes are logically organised into a *virtual ring*, ordered by their identifier. A node connects to its $r/2$ closest neighbours both clockwise and counter-clockwise. The resulting set of r closest neighbours a node is connected to is called the virtual neighbour set, short *vset*. A typical *vset* size is $r = 4$.

Figure 1 on the left shows an example virtual ring using this *vset* size and 12-bit identifiers as well as the *vset* of node 8F6. On the right it is demonstrated that the members of a *vset* are randomly distributed across the physical topology of the network. This is due to the identifiers being randomly assigned, as mentioned above.

Besides the *vset*, a node also maintains a set of closest neighbours in the physical network topology, called the physical neighbour set, short *pset*.

2.2 Routing

Routing in VRR provides a DHT, that is, lacking an explicit route to the destination, messages are routed to the node that has the identifier numerically closest to that of the destination node.

Each node maintains routing paths (routes) to all other nodes in its *vset*, called *vset-paths*. As each node is in the

vset of each of the nodes in its *vset*, *vset-paths* are bidirectional. In Figure 1 the *vset-paths* of node 8F6 are visualised by spirals.

The *vset-path* is stored in the routing table of every node along that *vset-path*. That means, that besides its own *vset-paths*, the routing table of a node also contains all *vset-paths* that are routed through it.

Furthermore, the routing table contains physical neighbour paths to all the nodes in its *pset*. In addition to these directly reachable neighbours, as an optimisation, physical neighbour paths to its two-hop neighbours, i. e. the neighbours of its neighbours, are also stored in the routing table of a node. How these two-hop physical neighbour paths are populated, will be described later in Section 2.3.

Each entry in the routing table contains the identifiers of both endpoints of the path, endpoint_A and endpoint_B, the identifiers of the neighbour which is the next hop towards each endpoint, next_A and next_B, the identifier of the two-hop neighbour which is the second hop towards endpoint_A, nextnext_A and finally a *vset-path* identifier, path id.

Endpoint_A is the originating endpoint of the *vset-path*, i. e. the node that initiated the *vset-path*. The identifier in the next hop entries next_A and next_B obviously have to be members of the *pset* of the node. The two-hop neighbour nextnext_A is used for local path repair, as described later in Section 2.5.

The path identifier is an 8-bit unsigned integer value. For *vset-paths* it is unique per originating endpoint. Thus the tuple $\langle$path id,$\text{endpoint}_A\rangle$ identifies a *vset-path* globally unique. Physical neighbour paths always use the highest possible path identifier value, i. e. FF. The reason for this is, that the path identifier is used to select among multiple routes to the same destination, choosing the one with the highest path identifier. Using the highest possible value therefore ensures that routes to directly reachable physical neighbours are always preferred.

endpoint_A	endpoint_B	next_A	next_B	nextnext_A	path id
8F6	8F0	NULL	20E	NULL	03
8F6	8E2	NULL	F01	NULL	2F
90E	8F6	7E2	NULL	A3C	1E
910	8F6	F01	NULL	6D5	2F
35F	37A	20E	7E2	2A3	12
A01	A10	F01	FC1	6D5	F0
8F6	20E	NULL	20E	NULL	FF
8F6	F01	NULL	F01	NULL	FF
8F6	7E2	NULL	7E2	NULL	FF
8F6	FC1	NULL	FC1	NULL	FF

Figure 2: A sample routing table

Figure 2 shows part of the routing table for node 8F6 from previous examples. The first four entries consist of the *vset-paths* between 8F6 and the members of its *vset*. For the first two entries, 8F6 is the originating node of the *vset-path*, endpoint_A. As it is endpoint_A itself, logically, the next hop identifiers towards that endpoint, next_A and nextnext_A, are set to NULL.

The fifth and sixth entries are *vset-paths* that are routed through 8F6. The last four entries are the physical neighbour paths from 8F6 to the members of its *pset*.

The forwarding decision in VRR works as follows: VRR chooses the endpoint with the numerically closest identifier to that of the destination. If that endpoint is the node itself, the message is delivered locally, otherwise the message is forwarded to the next hop towards the selected endpoint, as indicated by the routing table entry. If there are multiple alternative routes to the destination, VRR, as aforementioned, selects the one with the highest path identifier.

2.3 Physical neighbours

To detect link failures quickly, nodes broadcast *hello* messages to their physical neighbours in a regular interval of T_h seconds. A typical value for that interval is $T_h = 1$ second.

A node maintains a set of those nodes, that it received a *hello* message from during the last $2kT_h$ seconds, with a typical value of $k = 4$.

A physical neighbour of a node can be in one of four states. A node not in the aforementioned set is in the *unknown* state. A neighbour that is in the set, i. e. from which *hello* messages were received, and that also received *hello* messages from the node, is in the *linked* state. The set of neighbours in the *linked* state is what forms the *pset* of the node. If the node does not know whether a neighbour, from which *hello* messages were received, also received *hello* messages from the node, that neighbour is in the *pending* state. If a link to a neighbour is marked as faulty, that neighbour is in the *failed* state.

To facilitate classification of the neighbours into these states, a node sends within each *hello* message the set of *linked* active nodes, the set of *linked* non-active nodes and the set of *pending* nodes. When a node sees itself in one of these sets, it knows that the neighbour has received its *hello* messages. Upon reception of a *hello* message from a neighbour, a node updates the state of the neighbour based on the neighbours old state and its own presence in the sets sent within the *hello* message. It can be represented as three states, *missing* if the node is not present in any of the sets, *pending* if it is in the set of *pending* nodes and *linked* if it is in either of the sets of *linked* nodes.

This leads to the state transitions illustrated in Figure 3, where nodes of the graph represent the state the neighbour is in and the edges represent the state of the node, as indicated by the neighbour's *hello* message.

When a node did not receive a *hello* message from a neighbour during the last kT_h seconds, it marks that neighbour as *failed*. If no *hello* message is received for another kT_h seconds, the neighbour enters the *unknown* state and is purged from the set of nodes that sent a *hello* message. This results in the aforementioned period of $2kT_h$ seconds, that a neighbour is remembered in that set.

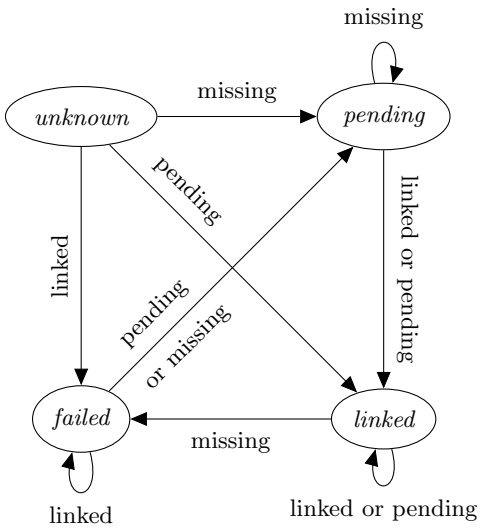

Figure 3: State transitions of physical neighbours.

Furthermore, information from received *hello* messages is used to update the routing table. When a neighbour is in the *linked* state and marked as active in its *hello* message, it belongs to the *pset* of the node and a routing table entry with this physical neighbour path is added, if not present.

From the set of *linked* active neighbours of the neighbour, a node is able to deduce the two-hop physical neighbour paths that are also added to the routing table.

2.4 Node join

A node joining the network first sets up its *pset* using *hello* messages as described in Section 2.3.

It then uses one of the active nodes in its *pset* as a proxy to setup its *vset-paths*. It first locates one member of its *vset*. This works by sending a *setup_req* message via the proxy to its own identifier. Due to the DHT nature of VRR, this message gets routed to a node whose identifier is numerically closest to that of the joining node, which of course is a node that belongs into the joining node's *vset*.

When the *setup_req* message reaches its destination, that node adds the joining node to its *vset* and, if that succeeds, answers with a *setup* message, else with a *setup_fail* message. Failure can happen due to concurrent joins, where a joining node is farther away from the node than other new node(s) that just joined, and the joining node thus happens to be outside of the *vset*.

The *setup* and *setup_fail* messages are sent back to the joining node via the proxy. If they, however, pass another physical neighbour of the joining node before the proxy, that neighbour will take the shortcut and route the message directly to the joining node. This is important, as each node the *setup* message passes through adds an entry for the *vset-path* originating from the joining node to the node sending the *setup* message to its routing table. This way the *setup* message sets up the *vset-path* and above mentioned shortcut ensures that the path goes via the closest neighbour instead of via the proxy.

Both, the *setup* and *setup_fail* messages also contain the *vset* of the sending node. The joining and, in case of the *setup* message, all nodes along the newly setup *vset-path* learn other potential *vset* members from this. Whenever a node learns about a node that belongs in its *vset*, but is not in it yet, it sends a *setup_req* message to it. As the first *setup_req* message was sent to the node whose identifier is closest to that of the joining node, the *vset* contained in the *setup* message sent back from that node contains all other *vset* members of the joining node, i. e. — assuming no concurrent joins that affect its *vset* — the joining node learns all the information it needs to complete the join from the first *setup* message received.

Setup of the *vset-path* may fail at any node among the path, the *setup* message is routed through, as aforementioned due to concurrent joins, as well as due to other failures that cause *setup* messages from nodes that are not in the *pset* or for already existing *vset-paths* to be received. As the *vset-path* is already partly setup at this point, it has to be destroyed — or torn down — to maintain consistent routing state. The procedure to facilitate this is called teardown and works by the node initiating it — in this case the node at which the *vset-path* failed — sending a *teardown* message towards the source of the *setup* message.

A node that receives a *teardown* message for a given *vset-path* removes that *vset-path* from its routing table. If it is the endpoint itself, it also removes the other endpoint from its *vset* and sends a *setup_req* message to that endpoint, else it passes it on to the next hop.

Teardown is also initiated for each *vset-path* to a node, whenever that node is removed from the *vset* due to another node with a numerically closer identifier joining the *vset*. As the *teardown* message — like the *setup* and *setup_fail* messages — also contains the *vset* of the sending node, the removed node learns about the joined node and can also update its *vset* accordingly.

After successful setup of *vset-paths* to all members of its *vset*, a node has successfully joined the virtual ring and becomes active.

The *setup* message also contains a *prev* field in which the identifier of the previous hop it was sent through is recorded. This is used to fill the nextnext_A item of the routing table entries.

A node that cannot find any active neighbours after a timeout, i. e. that ends up with an empty *pset*, creates its own virtual ring. Once a node which is member of another virtual ring comes into range, the two rings will be joined using the mechanism described in Section 2.5.1.

2.5 Failure detection and repair

VRR uses acknowledgements and retransmissions for all messages except *hello* ones on a per-hop basis. These are used in addition to failures to receive *hello* messages, as described in Section 2.3, to detect link failure and mark a neighbour

as *failed*.

The physical neighbour states and the transitions between them, as described in Section 2.3, ensure that failure detection of physical neighbours is symmetric, i. e. both nodes will detect that the link between them failed within a relatively short time.

To also assure *symmetric failure detection* for *vset-paths*, a node that marks its neighbour as *failed* also tears down any *vset-paths* found in its routing table that have the failed neighbour as next hop. Due to the *symmetric failure detection* of physical neighbours this happens on both ends of the broken *vset-path* thus making the *vset-path* failure detection symmetric, too.

If, during teardown, any node on the path fails to acknowledge a *teardown* message after several retransmissions, it is marked as *failed*, consequently triggering further *teardown* messages for all paths routed through it. This assures consistent routing state even when complex failures occur.

Whenever a *vset-path* is torn down, the endpoint receiving the *teardown* message sends a *setup_req* message to the other endpoint of the *vset-path* (cf. Section 2.4). Besides repairing the *vset-path* to that node after a failure of one of the intermediate nodes, this also repairs failure of that other endpoint itself. This is so due to the DHT nature of VRR — if the other endpoint is dead, the *setup_req* message will be delivered to another node whose identifier is numerically closest to that of the dead node. This node is either a suitable replacement *vset* member or it already is a *vset* member. In the latter case it almost always has a suitable replacement *vset* member in its *vset*, which it sends along with its *setup_fail* message. In the rare case that it does not, the original node repeats the join process until successful.

Besides this global *vset-path* repair that involves the teardown of the whole *vset-path*, VRR also has a much cheaper — constant cost — *local repair* mechanism. *Local repair* works by replacing a failed link to some node with an available alternative route to that node. This only affects and involves the nodes between which the link failed, it is thus local.

The strategy for *local repair* works as follows: First, the node detecting a link failure looks up every *vset-path* which has the failed node as its next_A. For each of these *vset-paths* it then first checks whether its endpoint_A is a physical neighbour, if so it changes the entry to directly point to that endpoint, thus not only repairing, but also shortening the *vset-path*. Secondly, it makes use of the nextnext_A item of the entry (cf. Section 2.2), if that node is a physical neighbour it likewise changes the entry to use that node as next_A, again both repairing and shortening the *vset-path*. Failing these optimisations, the node next searches for a route to nextnext_A, leveraging the fact that its routing table also contains routes to all two hop neighbours. If such a node is found, the *vset-path* is repaired by changing the routing table entry to use the next hop of the alternative route as next_A. The length of the *vset-path* does not change in this case.

For those *vset-paths* that have the failed node as their next_B, the node simply delays the teardown process for $(k+1)T_h+\delta t$ seconds to allow for failure detection and *local repair* from the other side to happen. If that happens, the node aborts the delayed teardown process, otherwise the teardown leads to a normal *vset-path* repair.

2.5.1 Network partitions

Another issue VRR has to deal with and repair is a partitioned network. When the network becomes partitioned, each partition forms a virtual ring of its own. The mechanism to join two partitions, merging both separate rings into one, works using *representatives*. The *representative* of a virtual ring is the node, whose identifier is numerically closest to zero. A *representative* obviously only has nodes with numerically larger identifiers than its own in its *vset*. Thus each node can decide locally, whether it is a *representative*, or not.

Every active node maintains a path to each *representative* it knows of in its routing table. Updates of routes to at most the two *representatives* whose identifiers are numerically closest to zero are sent within aforementioned *hello* messages. When through this, a node learns about a *representative* which should be in its *vset*, it sends a *setup* message to it. The fact, that the node learned of it through the routing updates and that each node maintains routes to each *representative* assures that the *setup* message can be routed to the *representative* across the partition into the other ring.

The *vset-path* setup initiated by this causes the two rings to merge at this point. The nodes from each ring that are removed from the *vset* of their former virtual ring neighbours due to nodes from the other ring taking their places, setup new *vset-paths* to their virtual ring neighbours in the merged virtual ring — thereby completing the merge of both rings.

3. ROUTING ON FLAT LABELS

3.1 Overview

Like VRR, Routing on Flat Labels (ROFL) uses unique, location independent node identifiers. They, however, are furthermore self-certifying, i. e. they are the cryptographic hash of the public key of a public-private key pair, thus allowing nodes to prove their identity. As an extension to implement anycast and multicast, ROFL both allows one node to hold multiple identities as well as multiple nodes to share an identity.

Again similar to VRR, nodes are logically organised into rings, ordered by their identifier. There are, however, multiple rings, nodes are members of an *internal* ring for intra-domain routing within an autonomous system (AS) and *external* rings for inter-domain routing to other ASes.

In each ring, a node has both a logical *predecessor* and *successor*. A node, however, only maintains pointers — also called *fingers* — to its *successor*. As an optimisation for better resilience, a node may maintain multiple *successor* pointers within a ring, called *successor-groups*.

The inter-domain routing in the Internet is governed by the hierarchical provider-subscriber relationships between the

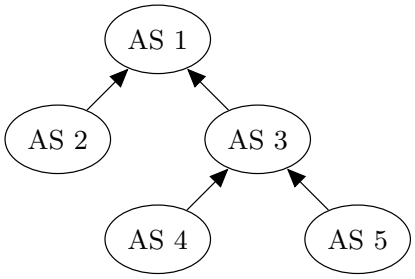

Figure 4: Sample AS hierarchy.

involved autonomous systems. Figure 4 shows such an example AS hierarchy, the edges being connections between the ASes and their direction denoting the subscriber-provider relationship.

ROFL uses the notion of an *up-hierarchy*, which is the upstream hierarchy as seen by a router. Figure 5 shows the resulting *up-hierarchy* as seen by the routers of AS 4.

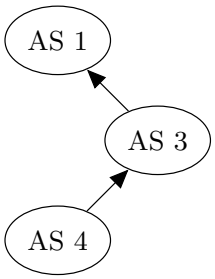

Figure 5: *Up-hierarchy* of AS 4.

Figure 6 shows the *internal* rings of AS 4, AS 5 and their upstream provider AS 3. It also shows sample internal *successor* pointers in each AS as well as sample external *successor* pointers from and to a host in AS 4. The setup of external *successor* pointers will be described in detail in Section 3.3.

As ROFL is designed for use in the Internet, not every host can reasonably be a node actively participating in the routing. ROFL thus distinguishes between three types of nodes: *routers* which do the actual ROFL routing, *stable hosts* which are hosts that are permanently connected to the network and *ephemeral hosts* which are hosts that are only intermittently connected or mobile.

Hosts connect to the ROFL network via *routers*. The *router* a host connects to is called the *hosting router* of the host and the host is called a *resident* host at this router.

Thus a typical site network as used today would be translated as follows into the ROFL scheme: The servers and some always online workstations would be *stable hosts*, workstations that only run during office-hours and laptops would be *ephemeral hosts*, routers that route between LAN segments and the site backbone would be *hosting routers* of the hosts in the connected LANs, and finally the border routers would also be border routers.

3.2 Routing

Routing in ROFL also provides a DHT, packets are routed to the node whose identifier is numerically closest and lower or equal to the identifier of the destination.

A routing path in ROFL is called a *source route*, it is a hop-by-hop list of router identifiers along the physical path.

To detect and identify link — and thus *source route* — failures, ROFL requires a routing protocol on the link level that maintains a *network map*, like OSPF [12] does. Intra-domain that routing protocol finds and maintains routes between hosting routers, inter-domain it maintains routes between border routers of interconnected ASes.

As mentioned above, each node has a *predecessor* and *successor* and it maintains *successor* pointers to nodes in each ring it participates in. However, *ephemeral hosts* are an exception to that. They are not part of any ring itself, they are neither *predecessor* nor *successor* to another node. They rather are "leaf nodes" to their *internal* ring, as illustrated in Figure 6. The node that would be — in terms of identifier order — the *predecessor* of the *ephemeral host* maintains a *source route* to the *ephemeral host*. Due to the DHT nature of the routing, any packet destined for the *ephemeral host* will reach the *predecessor* node which then can route it to the *ephemeral host*.

A node has *successor* pointers to an *external* ring, only when the *successor* in the *external* ring would be the direct *successor* of the node if both rings were merged, i. e. the identifier of the *successor* in the *external* ring is numerically closer to the node than that of its *successor* in the *internal* ring.

To reduce stretch, nodes additionally have *proximity-based fingers* to nodes that are close in the physical network topology. As another optimisation, routers have a so called *pointer cache* where they cache *source routes* that run across them. They, however, only cache pointers to a node, if they either are the *hosting router* of a *predecessor* of that node or if they lie on the shortest path to such a router. This restriction is required to allow for efficient cache invalidation on host failure, as later discussed in Section 3.4.

The forwarding decision of a router in ROFL works like this: The router first looks for the best match among its *resident virtual nodes*[1]. If it finds a direct match, it delivers the packet to that node, otherwise it looks for the best match in its *pointer cache* and forwards the packet to the better — i. e. closer to the destination — of these two matches.

ROFL maintains what it calls the *isolation property*. This property means, that a packet routed between two ASes will never travel higher in the hierarchy than through their least-common ancestor. For example, in the sample hierarchy illustrated in Figure 5, the *isolation property* would guarantee that a packet routed from AS 5 to AS 4 would never travel on a higher hierarchy level than through AS 3. Border routers that optionally use their *pointer cache* for inter-domain routing too, only use them if the destination

[1]See Section 3.3. For now, it is sufficient to read that as *resident hosts*, the only important part is that this lookup happens local.

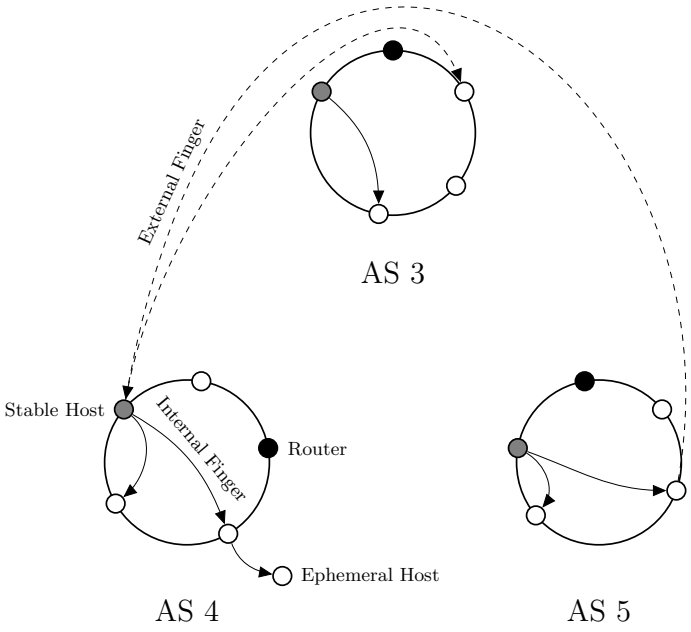

Figure 6: *Internal* and *external* rings.

host is at a higher level in the hierarchy. This is so to assure that the *isolation property* is not violated when for example the shorter route in the *pointer cache* leads to a path through a higher level AS. The decision of whether a host is at a higher or lower level of the hierarchy is facilitated by matching the host against a bloom filter [2] that contains the — known — (sub)set of hosts that are at a lower level of the hierarchy.

3.3 Node join

A host joins the ROFL network by contacting its *hosting router*. It locates it the same way hosts locate their default gateway in TCP/IP, i. e. by DHCP and other autoconfiguration means or static configuration. The host first proves its identity to the router using its self-certifying identifier. If that succeeds and the host is authorised to join the network, the router initiates the join process. This works by first creating a *virtual node* on behalf of the joining host. Next, the router finds the *predecessor* of the host in the *internal* ring, i. e. the node whose identifier is numerically closest, but lower than that of the joining node. Now the router inserts the *virtual node* into the ring by setting its *successor* to the current *successor* of the *predecessor* and then setting the *successor* of the *predecessor* to the *virtual node*.

The router then joins the *virtual node* into the *external* rings in its *up-hierarchy*. Therefore it selects — based on eventual policy requirements of its AS and the joining host — a set of paths along its *up-hierarchy* on which to potentially join *external* rings at each level. For each of these paths, the router forwards the join request including the path to a border router of the next hop AS on the path.

A border router receiving such an external join request, initiates the external join process for its ring. During this, it first finds the *virtual node* within the subtree of the hierarchy rooted at the current level that would be the *predecessor* of the joining *virtual node*. It then gathers both the sets of all — *internal* and *external* — *successors* of that "would-be *predecessor*" and of the joining *virtual node*. Both sets are also limited to those *virtual nodes* that are within the subtree of the hierarchy rooted at the current level.

Next the *virtual nodes* with the lowest identifier — i. e. the closest *successors* — of each set are compared. If that of the "would-be *predecessor*" would be a better *successor* than that of the joining *virtual node*, that *successor* is added as an additional external *successor* to the joining *virtual node*. Similarly, if the joining *virtual node* itself would be a better *successor* to the "would-be *predecessor*" than the best of its current *successors*, it is added as an external *successor* to the "would-be *predecessor*".

This scheme makes sure that an external *successor* at a given level of the *up-hierarchy* is always closer than any other *successor* — *internal* and *external* — at a lower level. An external *successor* at a higher level that would be farther away than an external *successor* at a lower level, could be reached by routing to that lower level external *successor* anyway and not doing so, i. e. not routing through the lowest possible AS in the *up-hierarchy* could violate the *isolation property*. Thus this scheme is necessary to ensure that this property is maintained.

Finally, unless being itself at the top level of the *up-hierarchy*, the router passes the external join request on to a border router of the next higher AS in the path supplied with the request. Thus the external join process recursively continues

through all levels of the *up-hierarchy* on the path specified by the *hosting router* of the joining host.

To bootstrap itself, a router after startup first creates a *default virtual node* with its own identifier. The router then joins this node into the *internal* ring by flooding a message announcing its identifier. Upon reception of this message, its *predecessor* links itself to the joining router and its *successor* responds to the message with his identifier, so that the joining node can link to it.

3.4 Failure detection and repair

The aim of ROFL is to ensure reachability of any two nodes between which a working network path exists and to ensure that for each of its pointers, a node detects the failure of the path to the destination or of the destination itself and consequently deletes that pointer.

Host and router failures are detected through session timeouts, link failures are detected by the underlying routing protocol on the link level.

When a host fails not only the *hosting router* is affected, but also any other router that has pointers to the failed node. Aside from the *hosting routers* that have *virtual nodes* whose *successor* pointers point to the failed host and thus have to be updated, in theory this could be any router because of the *pointer cache*. But due to the restriction of what a router may cache (cf. Section 3.2), the set of affected routers whose *pointer cache* may have to be invalidated is limited. The nature of that restriction, i. e. the limitation to routers that are the *hosting router* of the host's *predecessor* and those that lie on the shortest path to the former, includes the routers affected in any case in the former group, and those on the path to these in the latter. Therefore, the *hosting router* can easily send a source-routed flood that only reaches this set of affected routers, a so called *directed flood*, to inform them about the failure.

To facilitate efficient failover in case of router failures, routers agree upon a sorted list of routers to fail over to. When a router fails, the next alive router on that list is the failover router to be used instead.

Each node that was *resident* at the failed router rejoins the network using the failover router as soon as it detects the failure. To provide higher resilience, hosts may instead join the ROFL network using multiple *hosting routers* right away, so that they stay connected when a router fails.

Besides the *resident* hosts, *hosting routers* of hosts that have *successor* pointers to the *virtual nodes* at the failed router are also affected. Due to the deterministic failover router selection, these routers can easily adjust the affected *successor* pointers to the failover router.

When a link fails without creating a partition of the *internal* ring, all a router needs to do is to invalidate *pointer cache* entries whose paths contain the failed link. The network map maintained by the underlying routing protocol on the link level takes care of discovering alternative paths.

However, when a partition of the *internal* ring occurs, first, the separate partitions must form consistent rings of their own and then, as soon as the physical network link between the partitions is restored, they must again merge into a single ring.

The first goal is achieved by any routers detecting pointers that became invalid due to the partition repairing them locally. This works by pointing an invalid *successor* pointer to the *resident virtual node* with the numerically closest, but higher identifier, i. e. the next available identifier.

This basically "closes the gaps" the missing nodes from the other partition left behind.

Merging ROFL rings works similar to merging in VRR as described in Section 2.5.1. Just like the *representative* in VRR, routers in ROFL determine the *zero-ID*, which is the smallest, i. e. closest to zero, identifier they know. They distribute the *zero-ID* along with link state advertisements of the underlying routing protocol on the link level. When network connectivity between the partitions is restored and thus the node in one partition that should be the *predecessor* of the *zero-ID* in the other partition learns about the presence of the *zero-ID*, it will link its *successor* pointer to it. This triggers the merge process, wherein each node in turn repairs its *successor* resulting in a merged ring.

To make sure that the *internal* rings at each level of the hierarchy merge consistently, routers also maintain a route to the *zero-ID* of the hierarchy levels below them.

3.5 Multi-homing and peering

Besides the simple provider-subscriber relationship assumed in the description of ROFL so far, there exist more complex constellations.

One of the simpler ones is a site subscribing to more than one provider without being multi-homed. To achieve this setup, a site simply joins different identifiers to the different providers.

The other simple case is that of a site being multi-homed to the same upstream AS — being it multiple connections to the same provider or connections to multiple providers that are no AS on their own and themselves subscribe to the same provider. This setup is simply a matter of applying policy to link selection towards the upstream AS.

The — from an availability standpoint — most desirable setup is multi-homing to different upstream ASes, as illustrated in Figure 7. AS 4 is multi-homed to both AS 2 and AS 3 and thus a single failure of either AS 2 or AS 3 will not affect its connectivity.

In ROFL this — in the current Internet more involved — setup also works quite simple. The *up-hierarchy* of the multi-homed AS contains all its upstream providers. Figure 8 shows the *up-hierarchy* of the multi-homed AS 4 from this example.

As long as no policy requirements prevent this, the join process (cf. Section 3.3) selects all *up-hierarchy* paths to initiate external join processes. Therefore, routers at a multi-homed

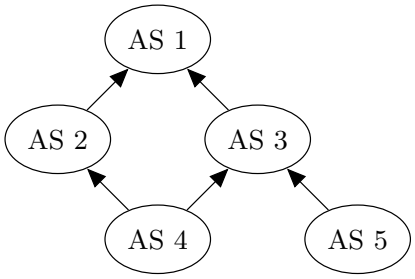

Figure 7: Multi-homing to two ASes.

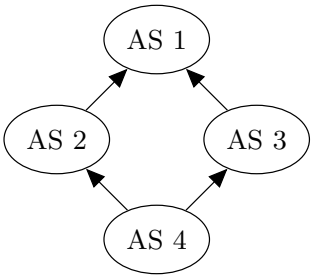

Figure 8: *Up-hierarchy* of AS 4, multi-homed case.

site will automatically join *external* rings of all its upstream providers, if applicable.

Usage of a link merely as backup link is achieved by according policies for the *up-hierarchy* path selection during the join process (cf. Section 3.3) that result in paths utilising the backup link to never be selected as long as the main link is up.

A preliminary extension to ROFL to provide better support for routing policies and traffic engineering adds optional suffixes to identifiers. A *hosting routers* of a multi-homed site appends a different suffix to the identifier in an external join request for each provider. Other hosts sending packets to hosts at the mulithomed site by default choose a random suffix which results in load balancing among the connections of the multi-homed site. Other hosts and routers, however, may as well choose specific suffixes to control which path the packets are routed along.

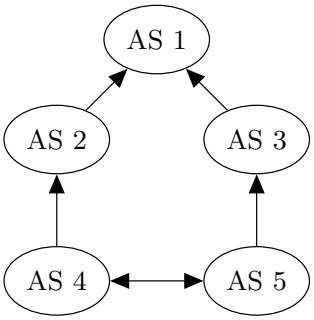

Figure 9: Peering between two ASes.

Peering of two ASes, in the sense of routing packets destined to each others AS over a direct peering link, as depicted between AS 4 and AS 5 in Figure 9, can be handled two different ways in ROFL.

The first way is to assume a *virtual AS* for each peering link as shown in Figure 10. The *virtual AS* is a provider to all peers and a subscriber to each peer's providers. Again, the *virtual AS* is just assumed, it only exists in the form of additional rules in the join process. A *virtual node* joins the *virtual AS* in such a way that it will join the other peer's *internal* ring, but not that of the other peer's upstream provider. In the example in Figure 10, AS 4 may join the *internal* ring of AS 5 and vice versa, but neither may AS 4 join the *internal* ring of AS 3 nor may AS 5 join the *internal* ring of AS 2.

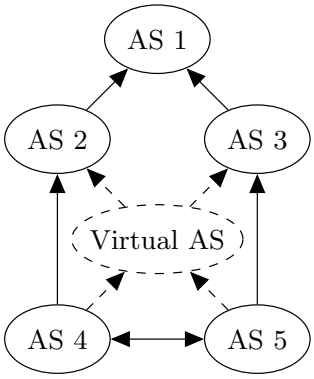

Figure 10: Peering using virtual AS.

The second way is to use bloom filters [2] to ensure that only packets destined to hosts *resident* at the peering AS are routed across the peering link. Each peer checks the destination identifier of a packet against the bloom filter for the other peer's hosts, before routing it to that peer. If it does route the packet, it marks it as being sent over a peering link. This is necessary to deal with false positives the bloom filter may match. These are sent back over the peering link by the other peer and will then — due to the mark — not be routed across the peering link again, but instead some other path.

The *virtual AS* approach has the advantage of not needing such a backtracking mechanism, the bloom filter approach, however, comes without the additional join overhead of the former.

3.6 Anycast and multicast

Anycast builds upon the ROFL extension used for policy routing and traffic engineering in a multi-homing setup described in Section 3.5. Servers wishing to be reached via an anycast identifier join the ROFL network using the identifier of the desired anycast group and a suffix. Routers forward packets destined to the identifier of the anycast group independent of the suffix. Thus these packets get routed to the closest — in the ROFL topology — server of the anycast group.

Multicast in turn builds upon anycast to facilitate joining a

multicast group. The host that wants to join sends a request message using anycast to the multicast group identifier, consequently reaching a close member of the multicast group. When this message passes through a router, the router sets up pointers to the multicast group in the direction the request message came from. Thus, when it reaches a multicast group member, a path for the multicast group identifier from that member to the joining host exists, i. e. the host has successfully joined the multicast group. Multicast packets sent to a multicast group are forwarded to all links of a router for which pointers to that multicast group identifier exist, except for the link the packet arrived at.

4. CONCLUSIONS

Both discussed routing protocols constitute a novel approach to routing. They leverage the advantages of Distributed Hash Table techniques for efficient, scalable routing that in addition to point-to-point routing provides DHT functionality. As another notable benefit, they do away with the constraints of hierarchical, location dependent addresses serving as both identifiers and locators. They do so by being able to route based on true, flat, location independent identifiers alone, without the need for any lookup or mapping mechanism between identifiers and locators.

VRR is designed as an efficient mesh routing protocol for wireless ad hoc networks. The design shows various optimisations for this scenario. The mixture of paths to both virtual and physical neighbours and even to two-hop physical neighbours along with the different small optimisations to take shortcuts in the routing help to reduce stretch. The *symmetric failure detection*, the short *hello* message interval and the optimisation of local path repair helps keeping the packet delivery rate high despite frequent failures of nodes. Low per-packet overhead and never flooding the network helps to preserve scarce bandwidth in wireless networks thus helping scalability.

According to the simulation and experimentation results presented in the original paper [3], VRR manages to realise these design properties and performs at least as good as or better than any of the mesh routing protocols it was compared to in various settings. Stretch is less than 1.2 for 50 nodes and slightly greater than 1.4 for 200 nodes.

ROFL is designed as a proof of concept, intra- and inter-domain routing protocol for use on the Internet, offering anycast as well as multicast, and — due to the location independent identifiers — host mobility. While routing on a flat hierarchy itself, it does take into account provider hierarchies including peering, multi-homing and routing policy requirements. It achieves this by combining and modifying ideas from different DHT algorithms and VRR, yet leaving implementation detail open and up to further experimentation in some parts. The simulation results presented in the original paper [4] show that acceptable performance is only reached by using sufficiently many *proximity-based fingers* and large *pointer caches*. Doubling the number of *fingers* to 680 and a *pointer cache* of 20 million entries is required to reduce average stretch from 2.5 to 1.33.

On the one hand this shows that ROFL merely is a proof of concept, on the other hand it shows that performance is surprisingly good for a first attempt at what before was thought of as being impossible — Internet routing without hierarchy.

While different in scope and objective, both routing protocols demonstrate that using Distributed Hash Table techniques as used in overlay networks for the routing in the underlay network is a worthwhile area of further research.

5. REFERENCES

[1] C. Adjih, E. Baccelli, T. H. Clausen, P. Jacquet, and G. Rodolakis. Fish eye OLSR scaling properties. *IEEE Journal of Communication and Networks (JCN)*, Special Issue on Mobile Ad Hoc Wireless Networks, Dec. 2004.

[2] B. H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, 1970.

[3] M. Caesar, M. Castro, E. B. Nightingale, G. O'Shea, and A. Rowstron. Virtual ring routing: network routing inspired by DHTs. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 351–362, New York, NY, USA, 2006. ACM Press.

[4] M. Caesar, T. Condie, J. Kannan, K. Lakshminarayanan, and I. Stoica. ROFL: routing on flat labels. In *SIGCOMM '06: Proceedings of the 2006 conference on Applications, technologies, architectures, and protocols for computer communications*, pages 363–374, New York, NY, USA, 2006. ACM Press.

[5] V. Fuller. Scaling of internet routing and addressing: past view, present reality, and possible futures. IEPG Meeeting, Nov. 2006. http://www.iepg.org/november2006/2-routing-scaling.pdf.

[6] V. Fuller and T. Li. Classless Inter-domain Routing (CIDR): The internet address assignment and aggregation plan. RFC 4632 (Best Current Practice), Aug. 2006.

[7] P. Ganesan, K. Gummadi, and H. Garcia-Molina. Canon in g major: Designing dhts with hierarchical structure. In *ICDCS '04: Proceedings of the 24th International Conference on Distributed Computing Systems (ICDCS'04)*, pages 263–272, Washington, DC, USA, 2004. IEEE Computer Society.

[8] R. Hinden, S. Deering, and E. Nordmark. IPv6 global unicast address format. RFC 3587 (Informational), Aug. 2003.

[9] R. Hinden, M. O'Dell, and S. Deering. An IPv6 aggregatable global unicast address format. RFC 2374 (Historic), July 1998. Obsoleted by RFC 3587.

[10] G. Huston. Commentary on inter-domain routing in the internet. RFC 3221 (Informational), Dec. 2001.

[11] G. Huston. The CIDR report, Dec. 2006. http://www.cidr-report.org/.

[12] J. Moy. OSPF version 2. RFC 2328 (Standard), Apr. 1998.

[13] J. Saltzer. On the naming and binding of network destinations. RFC 1498 (Informational), Aug. 1993.

[14] I. Stoica, R. Morris, D. Liben-Nowell, D. R. Karger, M. F. Kaashoek, F. Dabek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup protocol for internet applications. *IEEE/ACM Trans. Netw.*, 11(1):17–32, 2003.