To appear in: Proc. of the Third International Workshop on Hot Topics in Peer-to-Peer Systems, Rhodes Island, Greece, April 25-29, 2006

# Linyphi: An IPv6-Compatible Implementation of SSR

Pengfei Di, Massimiliano Marcon, and Thomas Fuhrmann
IBDS Systemarchitektur
Universität Karlsruhe (TH), Karlsruhe, Germany
{di,marcon,fuhrmann}@ira.uka.de

## Abstract

*Scalable Source Routing (SSR) is a self-organizing routing protocol designed for supporting peer-to-peer applications. It is especially suited for networks that do not have a well crafted structure, e. g. ad-hoc and mesh-networks. SSR is based on the combination of source routes and a virtual ring structure. This ring is used in a Chord-like manner to obtain source routes to destinations that are not yet in the respective router cache. This approach makes SSR more message efficient than flooding based ad-hoc routing protocols. Moreover, it directly provides the semantics of a structured routing overlay.*

*In this paper we present Linyphi, an implementation of SSR for wireless accesses routers. Linyphi combines IPv6 and SSR so that unmodified IPv6 hosts have transparent connectivity to both the Linyphi mesh network and the IPv4/v6 Internet. We give a basic outline of the implementation and demonstrate its suitability in real-world mesh network scenarios. Linyphi is available for download [1].*

## 1. Introduction

The Internet has faced an enormously growing popularity during the last decade. Meanwhile, more and more people are setting up private home networks. Often, these networks consist of WiFi clouds that do not only contain several notebook computers but also an increasing number of consumer electronic devices. With peer-to-peer applications it seems attractive to connect neighboring private networks directly, not only via a publicly operating Internet provider.

This increased and modified use confronts the Internet architecture with important challenges. A well-known challenge is the depletion of the IPv4 address space. It has been accounted for by the 128bit address space of IPv6. Another challenge is that of routing in such large unstructured networks, i. e. networks where laypeople connect their private network clouds arbitrarily. To our belief this latter problem has not yet been sufficiently accounted for.

In this paper we present a practical approach to resource efficiently support routing in such scenarios, the *Linyphi* mesh network router. Linyphi is an implementation of the recently proposed *scalable source routing* (SSR) protocol on a popular embedded DSL/WiFi home network router, the Linksys WRT54GS. SSR provides fully self-organizing, efficient routing in large unstructured networks. Externally, i. e. towards the Internet provider, Linyphi behaves like a regular DSL router. Internally, Linyphi acts as mixed IPv6 and SSR router. Among each other, the Linyphi routers use SSR to efficiently route the (encapsulated) IPv6 traffic.

With Linyphi, unmodified IPv6 capable hosts obtain full IPv6 connectivity, both to the Internet and to all other IPv6 hosts within the Linyphi mesh network. This is especially useful for peer-to-peer applications that need to transfer large amounts of data, but are not restricted to a particular server on the Internet. If a suitable peer is available in the local Linyphi mesh cloud, the traffic can be kept local.

Due to SSR's self-organizing properties, the Linyphi mesh network can grow very large. Moreover, local Linyphi clouds can be arbitrarily connected to other clouds without any further configuration. This can be done wired (Ethernet), wirelessly (Wave LAN), or implicitly by tunneling through the Internet. After some time, we expect the Linyphi cloud to cover e. g. whole cities, thereby providing a powerful network infrastructure for most kinds of applications, as well as peer-to-peer applications.

Let us illustrate this with the following example: When a user sets up her first Linyphi router, she obtains a private wireless LAN cloud that uses IPv6 internally, but otherwise provides the very same service as a regular off-the-shelf wireless LAN home router. When her neighbor sets up a Linyphi router, too, they can connect both routers to obtain a small Linyphi mesh cloud. Any traffic sent to destinations within this cloud is routed locally. This is especially useful for peer-to-peer applications where this can lead to a significantly increased performance. The more neighbors join the Linyphi cloud, the greater the performance increase. But even a small local cloud can be worth the little effort to use Linyphi instead of an off-the-shelf product.

We believe that this property, namely *no extra cost for installing the first device* and *increased performance from the second device on*, helps to quickly promote the deployment of Linyphi.

This paper is structured as follows: Section 2 summarizes the state of the art in ad-hoc and mesh routing and explains why those approaches are inefficient for peer-to-peer applications. Section 3 briefly introduces the scalable source routing protocol. (For details and a general performance evaluation the reader is referred to [4].) Section 4 describes the design of Linyphi, and section 5 extends this design to interoperability of the IPv4 Internet. Section 6 discusses the actual implementation on the Linksys WRT54GS and evaluates its performance. Section 7 concludes with an outlook to future work.

## 2 The Problems of Ad-Hoc Network Routing

The progress in wireless technology and the increasing popularity of ad-hoc networks with dynamic topology pose new challenges for the design of efficient routing algorithms. Traditional shortest path routing algorithms like Dijkstra and Bellman-Ford require proactive flooding of topology information through the whole network. Moreover, each node must be able to identify the next hop for all the destinations in the network. Consequently, in a $N$ nodes network where addresses are not assigned coherently, the routing table for each node will have $O(N)$ entries. This is not acceptable when the number of network nodes is large and the devices have limited memory capabilities.

On the other hand, hierarchical solutions, as applied e. g. in the Internet, are little suitable to grass-root dynamic ad-hoc networks with their frequent topology changes and lack of administration. For this latter type of networks, both reactive and proactive solutions have been proposed. But typically these solutions rely on flooding and are thus not scalable to large networks.

The *ad-hoc on-demand distance vector routing* (AODV) is perhaps the most widely discussed ad-hoc routing protocol [15] [2]. It floods the network with route request messages to find routes to previously unknown destinations. These routes are stored in the network so that the nodes know the direction to active sources and destinations. Dynamic source routing (DSR), too, uses flooding [12]. Unlike AODV it builds source routes between the active sources and destinations. These source routes are cached at the source and destination in a so-called route cache. [11] details this caching approach.

So far, there have been only few attempts to explore the potential interactions between overlay routing and such an ad-hoc routing protocol. Following an approach analogous to SSR, [9] and [10] propose an integration of DSR and Pastry [16] at the network layer of ad-hoc networks. There, packets are routed with the same algorithm as in Pastry, but each hop in the overlay network is a multi-hop source route. In their implementation however, an application needs to explicitly bind to an user-space library, while the infrastructure provided by Linyphi is completely transparent to its client hosts.

Other authors suspected that employing a structured routing overlay like e. g. Chord on top of a mobile network can lead to severe inefficiencies. As a consequence mobile peer-to-peer applications often combine infrastructure nodes with mobile nodes, or revert to unstructured solutions (see [6] and [5]). Scalable source routing (SSR) [4] is a recent approach to address these inefficiencies and combine the semantics of a structured routing overlay with an ad-hoc routing protocol.

In the following section we briefly describe the SSR protocol upon which we built Linyphi. SSR has recently been proposed as memory and message efficient routing protocol for large self-organizing networks. Typically, such networks will be meshes of wireless ad-hoc network clouds that are arbitrarily connected by both grass-root wire-line links and public network infrastructures.

## 3 Scalable Source Routing

SSR addresses the afore mentioned problems of routing in large scale ad-hoc networks by a combination of source routing with ideas from structured overlay routing, especially the Chord [17] routing overlay. As with Chord, SSR assumes every node to bear a globally unique identifier (SSR-ID) from a large circular identifier space. With SSR each node may randomly choose its ID, i. e. the node's position in identifier space is independent from the underlying network topology. (For Linyphi we derive the SSR-ID from the node's IEEE 802 MAC address; see section IV for details.)

When a node needs to send a packet to another node, it includes a source route *towards* the destination in the packet's header. This source route can be a list of the globally unique SSR identifiers of all the nodes that are to be traversed by the packet. It can also be a compact version giving only a list of locally unique peer identifiers (see [4] for details).

Optimally, the source node would have the full source route to the respective destination already in its route cache. This cache stores source routes to recently contacted nodes on a least recently used (LRU) basis. It can be efficiently organized into a tree-based data structure that allows quick access and has only a low memory footprint. Simulation studies [4] show that a cache size of about 250 nodes is sufficient in many cases. (Note that due to the small-world property of many networks, the required cache size scales with $O(\log \log N)$, i. e. it is in fact almost independent from the actual network size $N$.)

If the source node does not have the destination node in its cache, it can still prepend a source route *towards* the requested destination. To this end, the node employs the metric of the identifier space and looks up a node in its cache that is *virtually* closer to the destination than itself. Typically the cache will contain several such nodes. In such a case the node with the shortest source route is preferred, i. e. the node that is *physically* closest to the node that needs to prepend the source route. SSR calls such an intermediate node 'mediator'.

Obviously, the routing process can contain several such mediators. Nevertheless, it is guaranteed to converge if only each node has the source routes to its virtual neighbors in its cache. SSR takes special care to obtain and maintain these source routes to the virtual neighbors. This is done very efficiently, almost always without flooding.

The basic process is as follows: Each SSR node periodically broadcasts its SSR_ID to all of its physical neighbors. Upon reception of such a broadcast, an SSR node, let's call it $A$ for future reference, selects its two virtual neighbors, namely one successor $S(A)$ and one predecessor $P(A)$, among all the nodes $A$ knows of. Initially, the virtual neighbors have to be taken from the set of physical neighbors of $A$ since these are the only nodes known to $A$. Later they will be chosen from the entire route cache.

Now, $A$ sends a successor (or predecessor) notification message to $S(A)$ (or $P(A)$ respectively). Upon reception of such a message the receiving node, let's call it $B$ for future reference, checks whether it can detect an inconsistency. I. e. if $A$ thought $B$ to be its successor ($S(A) = B$), $B$ checks whether it agrees with $A$ being its predecessor ($P(B) = A$). If not, $B$ sends $A$ a neighbor update message, that informs $A$ about the node $C = P(B)$, i. e. the node that $B$ thinks to be its predecessor and which is thus a better candidate for $A$'s successor. By construction this message will contain the full source route from $A$ to $C$.

This process is performed similarly for all the mutual successor and predecessor relations until no node detects any further inconsistencies. It can be shown that in practice this process terminates rather quickly and reaches the globally consistent state. Furthermore, simulations show that SSR can keep the control message overhead very low, even in networks with churn and node mobility. Moreover, SSR scales well up to more than 100 000 nodes [4] with paths that are on average only about 20% longer than the globally optimal shortest paths. This small overhead is the price SSR has to pay for its memory and control message efficiency. However, in scenarios where a peer-to-peer application requires a structured routing overlay, e. g. Chord, this overlay also introduces a routing stretch [7].

This paper demonstrates SSR's applicability in a real world scenario. Making the structured routing overlay semantics accessible directly for a peer-to-peer application is among our current work.

## 4 Linyphi Design Issues

Linyphi's goal is to make SSR compatible with IPv6 such that SSR network clouds are transparent to normal end-hosts like e. g. Windows, Apple or Linux computers. To this end we differentiate between *routers* and *hosts*. A *router* is an intermediate node in the SSR network that communicates with other routers by means of the SSR protocol. Being a node in the SSR infrastructure, every router is provided with a unique SSR-ID. A *host* implements (only) IPv6. It is attached to at least one router which has to be its default IPv6 router. This router contains all the functionalities of a normal IPv6 router, except for the actual routing mechanism. It acts as a gateway to the SSR infrastructure. Note that a host and its associated router communicate only through standard IPv6, i. e. we *do not need to modify* a host for it to be able to join a Linyphi network.

### 4.1 Address space mapping

Since SSR and IP are different routing protocols they operate in different address spaces. By using IPv6's stateless address auto-configuration [18] we can very easily map the SSR address space into an unused part of the IPv6 address space. In particular, each router will broadcast a 64 bit IPv6 prefix (identifying the respective IPv6 subnet) on each of its interfaces.

Linyphi composes this prefix as follows: The first 8 bit of the prefix are set to 0100::/8, a still unused prefix for global uni-cast addresses [8]. The next 48 bit of the prefix contain the SSR-ID of the router. In our implementation this is the MAC address of one of the router's interfaces. The last 8 bit are set to the identifier of the router's interface from which the prefix is broadcast. (A Linyphi router can have up to 255 interfaces.) Summing up, the prefix of the Linyphi router advertisement is:

01:SSR-ID:Subnet::/64

It is important to note that by embedding the SSR-ID of the respective default router into the IPv6 prefix we can directly determine the SSR-ID of the destination router from the destination host's IPv6 address. To understand why this is important, consider the following example: When a source host sends a packet to some destination host it simply forwards this packet via IPv6 to its default router. This router (we call it the source router) encapsulates the IPv6 packet into an SSR packet. (In fact it prepends its SSR header to the IPv6 packet.) In order to forward the packet it needs to determine the SSR-ID of the *destination router*, i. e. the default router of the destination host. Given the afore mentioned property of the prefix, this can be done directly.

Since the identifier of the destination interface is also contained in the IPv6 address, processing at the destination router is further simplified. Nevertheless, for those packets, i. e. when the MAC address of the destination host is still unknown, the destination router needs to perform an IPv6 neighbor discovery before it can actually deliver the IPv6 packet to the destination host [13].

## 4.2 Packet Types and Formats

Linyphi distinguishes between three types of IPv6 packets: *on-link*, *global on-link* and *off-link*. A packet is *on-link* if the source and destination host are in the same subnet, i. e. source and destination address match with their full 64 bit prefix. A packet is *global on-link* when source and destination host are attached to the same router, but belong to different interfaces (i. e. subnets). In that case only the leading 56 bit of source and destination address match. Otherwise, a packet is *off-link*, i. e. source and destination router differ. Only in that case packets need to be encapsulated and forwarded as SSR packets via the SSR network cloud.

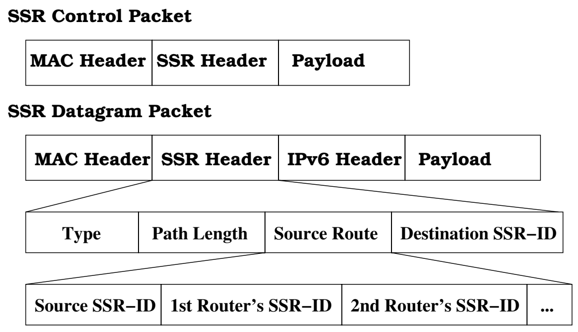

**Figure 1. SSR Packet Format**

Between Linyphi routers such SSR packets are sent in Ethernet frames with type code Ox8888. (Note that SSR needs its own type code to differentiate its packets from those of other routing protocols.) There are two major types of SSR packets: (1) datagram packets, containing an encapsulated IPv6 packet and (2) control messages required e. g. for maintaining the virtual ring. (See figure 1 for an illustration.) This differentiation corresponds to the difference between IP and ICMP packets.

As shown in figure 1, the general SSR header consists of four fields:

- Type (1 byte) indicating the type and subtype of the SSR packet.
- Path length (1 byte) giving the number of hops in the subsequent source route.
- Source Route ($6 * n$ bytes, where $n$ is the path length): A list of the SSR-IDs along which the packet will be sent hop by hop.
- Destination (6 bytes): The SSR-ID of the destination router. Note that this can differ from the last hop in the source route when the sender or a mediator node could not retrieve the full source route to the destination from its cache.

Depending on the control message type, the SSR header will contain further fields. (See the documentation accompanying the source code for details.)

## 4.3 Path MTU Discovery

As specified in [3], all IPv6 end-hosts have to support the so-called path-MTU discovery. If a packet is too long to be relayed further, the intermediate IPv6 router will send an ICMPv6 packet back to the sender containing the value of the MTU in which the packet should fit. The host will then resend an accordingly fragmented packet. Therefore, our Linyphi routers also have the functionality of IPv6 path-MTU discovery with small adjustment: Since a host is completely unaware of Linyphi, and has no knowledge of the overhead introduced by the SSR header, Linyphi must reduce the MTU value by L, the size of SSR header, before sending back the ICMPv6 packet.

## 5 Extension to IPv4 Internet

Although Linyphi aims primarily at providing IPv6 within a mesh network of IPv6 hosts, it is also capable of inter-operating with the IPv4 Internet. To this end Linyphi adopts the embedding of IPv4 addresses into IPv6 addresses and – more importantly – the mapping of IPv6 addresses to IPv4 addresses by the help of network address translation (NAT). Thereby, Linyphi routers can act (separately or jointly) as gateways to the IPv4 Internet.

### 5.1 NAT Gateways

Linyphi's NAT functionality is based on an IPv4-to-IPv6 translation mechanism that has been specified in [19]. Its core idea is to send packets that are destined to the IPv4 Internet via SSR datagram messages to a gateway that can translate the IPv6 packets into IPv4 packets, and vice versa.

Since each SSR router needs to be able to determine a gateway, each Linyphi router is provided with a gateway table, i. e. a list of the SSR-IDs of the Linyphi gateways that are available to the respective router. When an off-network packet arrives at the source router it chooses a gateway from the table and encapsulates the packet so that it can be forwarded via SSR to that gateway. The gateway will decapsulate the packet, translate it into an IPv4 packet, and store the respective state required for translating potential IPv4 traffic destined back into the Linyphi network.

Note that the NAT mechanism itself (performed by the gateways) is identical to the NAT as described in [19]. This also means that gateways need to treat NAT incompatible application layer protocols (e. g. DNS, FTP, etc.) by an additional NAT application layer gateway (ALG) mechanism. Our Linyphi implementation provides treatment for some popular protocols, but it is far from being exhaustive. Moreover, our implementation does not yet include cryptographic protection of gateways against unauthorized use. Hence Linyphi is currently only recommended for users with unrestricted flat rate Internet access.

### 5.2 Fragmentation Problem

As described above, path MTU discovery is mandatory for IPv6, but not for IPv4. Today, some IPv4 hosts perform path MTU discovery by setting the DF (don't fragment) flag in the IPv4 header. But others rely on the intermediate IPv4 routers to fragment the packets on their behalf.

In order to be interoperable with both types of IPv4 hosts, the Linyphi gateway fragments the too-long incoming IPv4 packets without a DF flag, or sends back an ICMPv4 packet indicating the appropriate MTU for those with the DF flag set (see [14]). Because of the varying length of the SSR header, each Linyphi router has the ability to perform IPv6 fragmentation, although they are encapsulated. Note that this is only utilized for those packets from the Internet.

### 5.3 DNS Service

Linyphi supports not only the use of DNS servers that reside within the Linyphi network. A host may also choose any DNS server from Internet using an IPv4 embedded IPv6 address format. Linyphi accomplishes this by a DNS ALG [19] on the gateway, performing DNSv4/6 translation.

## 6 Implementation and Evaluation

We used the Linksys wireless LAN router WRT54GS for our implementation of Linyphi. This is a popular Linux based MIPS device with five 10/100 Ethernet ports and an IEEE 802.11g interface. For testing we also used a SuSE 9.3 Linux desktop and WindowsXP notebook. Note that Linyphi does not require any modifications of the latter devices. All the functionality is contained in the router.

The Linksys router was equipped with OpenWrt, a Linux distribution particularly for the WRT54G/S routers. The initial version of OpenWrt was based on the original Linksys code for the firmware of the WRT54G router. This code was released under the GNU General Public License (GPL). Meanwhile, OpenWrt has developed into a fully-featured Linux system with the 2.4.30 kernel.

### 6.1 Architectural Overview

Linyphi has been implemented in C++. There are two main modules inside a router: the SSR kernel and the InterfaceKeeper (cf. fig 2). The SSR kernel executes the core algorithms for the SSR protocol, e. g. maintenance of the virtual ring, discovery of source

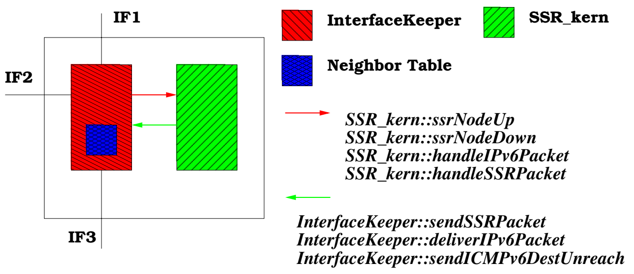
IF1
IF2
IF3
InterfaceKeeper
SSR_kern
Neighbor Table
SSR_kern::ssrNodeUp
SSR_kern::ssrNodeDown
SSR_kern::handleIPv6Packet
SSR_kern::handleSSRPacket
InterfaceKeeper::sendSSRPacket
InterfaceKeeper::deliverIPv6Packet
InterfaceKeeper::sendICMPv6DestUnreach

**Figure 2. Overview of a Linyphi router**

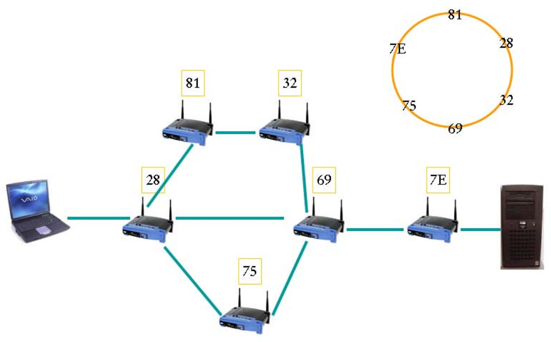

**Figure 3. Example with six Linksys routers**

routes, etc. . The InterfaceKeeper is responsible for the interaction with the Linux network stack, e. g. packet delivery and reception. Furthermore, it implements the required tasks of a normal IPv6router, like sending router advertisements, performing neighbor discovery, etc.

The interface between these two modules consists of the following seven methods:

- *void SSR_kern::ssrNodeUp(struct ssr_addr addr)*
  The InterfaceKeeper informs the SSR kernel of the presence of a newly attached SSR router.
- *void SSR_kern::ssrNodeDown(struct ssr_addr addr)*
  InterfaceKeeper informs the SSR kernel that a neighboring SSR router is no longer reachable.
- *void SSR_kern::handleIPv6Packet (struct ssr_addr addr, unsigned char *ipv6_packet, int len)*
  After reception of an IPv6 packet from an attached host, the InterfaceKeeper uses this method to pass the packet to the SSR kernel for encapsulation. Note that it provides the SSR address of the destination router, i. e. the router to which the packet's destination host is attached.
- *void SSR_kern::handleSSRPacket (const unsigned char* ssr_packet, int len)*
  After reception of an SSR message from a neighboring router, the InterfaceKeeper passes the packet to the SSR kernel for processing, including forwarding or decapsulation in case of datagram messages. In the latter case the SSR kernel will call *sendSSRPacket* or *deliverIPv6Packet* respectively.
- *void InterfaceKeeper::deliverIPv6Packet (const unsigned char* buf, int len)*
  When an SSR datagram message arrives at its destination router, the SSR kernel extracts the original IPv6 packet and calls this method to pass it to the InterfaceKeeper for delivery to the destination host.
- *void InterfaceKeeper::sendSSRPacket (struct ssr_addr addr, const unsigned char* buf, int len, int ip_offset)*
  The SSR kernel passes an SSR message to the InterfaceKeeper for forwarding it to the neighboring router identified by the given SSR-ID. Note that optionally the kernel can inform the InterfaceKeeper about the length of the SSR header. This is required for some IPv6 interoperability mechanisms, like changing the TTL value, sending ICMPv6 messages indicating the local MTU value, etc.
- *void InterfaceKeeper::sendICMPv6DestUnreach (const unsigned char* ipv6_packet, int len)*
  When the SSR kernel is unable to determine a route to the requested destination router, it calls this method. The InterfaceKeeper will then send an ICMPv6 destination unreachable packet back to the packet source.

The InterfaceKeeper also contains the so-called neighbor table that associates SSR-ID, MAC address and interface identifier of all neighboring Linyphi routers. With this table the InterfaceKeeper is able to send SSR messages via Ethernet frames when it is given the SSR-ID of the respective neighboring router. The InterfaceKeeper retrieves the SSR-ID of neighboring routers from their router advertisements. (As described above IPv6 router advertisements contain the SSR-ID of the respective router, thus they are also used as SSR hello messages in this implementation.) In order to detect the loss of a neighboring router, each entry in the neighbor table is associated with a timer that is reset upon reception of a router advertisement. When the timer expires, *ssrNodeDown* is called.

## 6.2 Experimental Evaluation

We tested our implementation with the network topology shown in fig 3. The notebook communicates with the server through a Linyphi network of six routers. The numbers depicted above each Linyphi router are the last bytes of the respective SSR-IDs. (Remember that Linyphi simply takes the MAC address of one of the router's network cards as the router's SSR-ID.)

About three seconds after the routers have been switched on, SSR has established its virtual ring, i. e. each router has found its virtual neighbors. (Note that this time corresponds to the time interval between SSR hello messages.)

When the notebook sends an IPv6 packet to the server, the default router of the notebook $R_{28}$ needs to find a source route to the default router of the server $R_{7E}$. At the beginning, $R_{28}$ does not know a path to $R_{7E}$. It therefore sends the packet to $R_{81}$, its predecessor on the ring. (Note that routing direction depends on virtual distance.)

Since $R_{7E}$ is a predecessor of $R_{81}$, the latter has cached a source route to $R_{7E}$, namely 81-28-69-7E. Using this source route, $R_{81}$ forwards the packet to $R_{28}$. Upon reception of the packet, the $R_{28}$ detects that there exists a source route from itself to $R_{7E}$. Hence it stores this new route 28-69-7E in its cache and subsequently uses this new route, which is actually the globally shortest. (Note that the latter is not always the case with SSR.)

When we break the link between $R_{28}$ and $R_{69}$, packets from $R_{28}$ to $R_{7E}$ are queued and finally dropped at $R_{28}$. After about four seconds the entry in the neighbor table has timed out and $R_{28}$ will delete the source route 28-69-7E. (Note that this delay is determined by the frequence of the IPv6 router advertisements. If $R_{28}$ knew that the link was broken and not only temporarily unavailable, it could immediately delete the respective source route.)

Once the broken source route has been deleted, $R_{28}$ sends the packets again to its predecessor $R_{81}$ who tries to forward it again via the source route 81-28-69-7E, i. e. sends it back to $R_{28}$. Upon reception of that packet $R_{28}$ immediately sends back a link broken message to $R_{81}$ who now regards $R_{7E}$ as unavailable. $R_{81}$ selects a new predecessor from its route cache, in our example $R_{32}$, and sends a predecessor notification. $R_{32}$ replies with a predecessor update, pointing $R_{81}$ to $R_{69}$. After the exchange of another notification-update pair, $R_{81}$ has again a source route to $R_{7E}$, namely 81-32-69-7E. The whole cascade from detection of the broken link to its repair takes in our setup less than 50 milliseconds.

| 3-hop (10Mb) | Ping6 | HTTP | FTP | SCP |
|---|---|---|---|---|
| Static Routing | 2.120 ms | 858 KB/s | 888 KB/s | 947 KB/s |
| SSR | 4.252 ms | 751 KB/s | 778 KB/s | 769 KB/s |

| 7-hop (10Mb) | Ping6 | HTTP | FTP | SCP |
|---|---|---|---|---|
| Static Routing | 4.571 ms | 865 KB/s | 909 KB/s | 912 KB/s |
| SSR | 10.94 ms | 664 KB/s | 687 KB/s | 665 KB/s |

|  | Ping | SCP (up) | SCP (down) |
|---|---|---|---|
| Home DSL | 48.7 ms | 60,6 KB/s | 292 KB/s |

**Table 1. Performance comparison between Linyphi and static IP routing**

Note that the packets now follow the source route 28-81-32-69-7E. The shortest path 28-75-69-7E has not (yet) been detected.

## 6.3 Performance Results

Although our implementation did not aim at high performance, we nevertheless compared Linyphi's performance to that of the unmodified Linksys routers. In particular, we tested four protocols (HTTP, FTP, SCP and Ping6) that can give a realistic impression of the performance as experienced by a Linyphi user. Table 1 gives the round trip time (RTT) and throughput results of that experiments. (The given numbers are average values of six runs each.)

The results show that Linyphi increases the RTT by almost a factor of 3, but the throughput is reduced only by about 10%–25%. Thus, we conclude that Linyphi is already useful for many bulk-data oriented peer-to-peer applications. This is especially true when we consider that with Linyphi peer-to-peer applications can keep their traffic in the local mesh cloud instead of always traversing e. g. a DSL connection. To illustrate that benefit, table 1 also gives typical values for a 6 MBit/s WiFi/DSL home access connecting to a machine at the local university.

We are confident that Linyphi's performance can be further increased in the near future. (Note that the current bottleneck of the Linyphi is the processor time.) One step is to integrate Linyphi into the routing subsystem of the Linux kernel. Especially packets already containing the full source route can thus be quickly processed on the "fast path". Another step is the reduction of the SSR overhead. This can be achieved by replacing the 6 byte SSR-IDs in the source routes with short labels that locally denote the next hop router. (See [4] for details.) With these improvements we expect Linyphi to achieve the performance of the regular Linksys router.

## 7 Conclusion and Future Work

In this paper, we presented Linyphi, a novel routing infrastructure that is based on the SSR protocol. Linyphi allows users to create large self-organizing meshes of wireless LAN clouds that are fully transparent to unmodified IPv6 hosts. Thereby, peer-to-peer applications running on these hosts have an increased likelihood to find peers locally, e. g. within the same block of flats, the same street, or the same neighborhood community.

We demonstrated that our current Linyphi implementation is within a factor of 1–2 of the performance of a state-of-the-art Linux router. Since the underlying routing protocol is self-organizing, Linyphi can be deployed even in large-scale settings almost without any configuration. Moreover, when compared to the currently most widely employed scenario where home networks are separately connected to the Internet via a DSL router, Linyphi already has a performance benefit of more than a factor of 2–3.

In the future, we expect to increase Linyphi's security and performance further. Concerning security we are investigating extensions to SSR that allow the introduction of SSR firewalls that can help to identify and filter malicious traffic. Such mechanisms would also support authentication, authorization and accounting, and e. g. control the use of the gateway connecting Linyphi to the Internet. Performance-wise we are currently optimizing the implementation, e. g. by streamlining the code, integrating a 'fast-path' into the kernel, and by reducing the protocol overhead in the packet header.

We are also planning an implementation that brings SSR directly to the host. Thereby, a host could benefit from SSR's support for node mobility, too. Moreover, since SSR can directly provide the semantics of a structured routing overlay, this would allow high performance mobile peer-to-peer applications.

## References

[1] www.linyphi.net.

[2] E. M. Belding-Royer and C. E. Perkins. Evolution and Future Directions of the Ad hoc On-Demand Distance Vector Routing Protocol. *Ad hoc Networks Journal*, 1(1):125–150, July 2003.

[3] S. Deering and R. Hinden. Internet Protocol, Version 6 (IPv6) Specification. RFC 2460, Dec 1998.

[4] T. Fuhrmann. Scalable routing for networked sensors and actuators. In *Proc. of the 2nd Annual IEEE Communications Society Conference on Sensor and Ad Hoc Communications and Networks*, Sept. 2005.

[5] I. Gruber, R. Schollmeier, and W. Kellerer. Performance evaluation of the mobile peer-to-peer service. In *2004 IEEE International Symposium on Cluster Computing and the Grid*, The Drake Hotel, Chicago, Illinois, USA, Apr. 2004.

[6] I. Gruber, R. Schollmeier, and F. Niethammer. Protocol for peer-to-peer networking in mobile environments. In *Proc. of the IEEE Workshop on Wireless Local Networks (WLN'03)*, Bonn, Germany, Oct. 2003.

[7] K. Gummadi, R. Gummadi, S. Gribble, S. Ratnasamy, S. Shenker, and I. Stoica. The Impact of DHT Routing Geometry on Resilience and Proximity. In *Proceedings of the SIGCOMM 2003 conference*, pages 381–394. ACM Press, 2003.

[8] R. Hinden and S. Deering. Internet protocol version 6 (IPv6) addressing architecture. RFC 3513, Dec 2003.

[9] Y. C. Hu, S. M. Das, and H. Pucha. Exploiting the synergy between peer-to-peer and mobile ad hoc networks. In *Proceedings of HotOS-IX: Ninth Workshop on Hot Topics in Operating Systems*, Lihue, Kauai, Hawaii, May 2003.

[10] Y. C. Hu, S. M. Das, and H. Pucha. Ekta: An efficient dht substrate for distributed applications in mobile ad hoc networks. In *Proceedings of the 6th IEEE Workshop on Mobile Computing Systems and Applications (WMCSA)*, English Lake District, UK, Dec. 2004.

[11] Y.-C. Hu and D. B. Johnson. Caching Strategies in On-Demand Routing Protocols for Wireless Ad Hoc Networks. In *Proceedings of the 6th Annual International Conference on Mobile Computing and Networking (MobiCom '00)*, pages 231–242, Boston, MA, USA, 2000.

[12] D. B. Johnson and D. A. Maltz. Dynamic Source Routing in Ad Hoc Wireless Networks. *Mobile Computing*, 353:153–181, Feb. 1996.

[13] T. Narten, E. Nordmark, and W. Simpson. Neighbor discovery for ip version 6 (IPv6). RFC 2461, Dec 1998.

[14] E. Nordmark. Stateless IP/ICMP Translation Algorithm (SIIT). RFC 2765, Feb 2000.

[15] C. E. Perkins and E. M. Royer. Ad hoc On-Demand Distance Vector Routing. In *Proceedings of the 2nd IEEE Workshop on Mobile Computing Systems and Applications*, pages 90–100, New Orleans, LA, USA, Feb. 1999.

[16] A. Rowstron and P. Druschel. Pastry: Scalable, distributed object location and routing for large-scale peer-to-peer systems. In *Proc. of the IFIP/ACM International Conference on Distributed Systems Platforms*, Heidelberg, Germany, Nov. 2001.

[17] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A Scalable Peer-to-peer Lookup Service for Internet Applications. In *Proceedings of the SIGCOMM 2001 conference*, pages 149–160. ACM Press, 2001.

[18] S. Thomson and T. Narten. IPv6 Stateless Address Autoconfiguration. RFC 2462, Dec 1998.

[19] G. Tsirtsis and P. Srisuresh. Network Address Translation - Protocol Translation (NAT-PT). RFC 2766, Feb 2000.