Computer Science

Georges Darakji and Jan Johansson

An Implementation of Distributed SIP for Wireless Mesh Networks

An Implementation of Distributed SIP for Wireless Mesh Networks

Georges Darakji and Jan Johansson

This report is submitted in partial fulfilment of the requirements for the Master's degree in Computer Science. All material in this report which is not my own work has been identified and no material is included for which a degree has previously been conferred.

Georges Darakji

Jan Johansson

Approved, 2008-06-04

Advisor: Andreas Kassler

Examiner: Donald Ross

Abstract

A mesh network is a self-healing, dynamic and usually wireless network with high bandwidth utilization, where nodes can freely move around within the topology, and where nodes completely independent of any centralized service.

Session Initiation Protocol (SIP) supplies a standardized way of establishing connections for many purposes, primarily Voice over Internet Protocol (VoIP) telephone calls. However, SIP relies a great deal on the existence of servers to locate other users. Servers are less appropriate to use on mesh networks. Neither does SIP does leverage any of its advantages; hence, it is not suitable providing a SIP service in a mesh network.

Our objective is to find and implement a concept which extends the functionality of the routers in a mesh network, thus minimizing the changes required on the end hosts.

Scalable Source Routing (SSR) is a routing approach which adds a Chord-like P2P overlay to any physical network such as a mesh network. Unlike other overlays (such as the ones created by P2PSIP), the routing mechanisms also considers physical distance which results in increased performance.

Linyphi is an implementation of SSR, which is designed to run on Linux based routers, such as the Linksys WRT54GL. SSR is used to route packets between the routers in a network, while IPv6 is used between the routers and the end hosts, thus enabling the end hosts to take advantage of SSR without any modifications.

In our solution, we will implement a SIP proxy which is to run together with Linyphi on the routers. To store the SIP addresses and locations of the SIP users, we will implement a DHT, which takes advantage of the SSR overlay.

Some other approaches were considered. The P2PSIP approach is a concept of replacing the SIP servers with a Distributed Hash Table (DHT). This DHT is contained within the SIP clients, thus no standard SIP clients can be used. This approach also adds a P2P overlay on

top of any other network, which causes performance of this approach to be dependent on the underlying layers. Using a P2P underlay would add even more overhead.

Skype uses flooding to find nearby users, which is very resource requiring, and it also uses a proprietary and closed protocol which prevents it from being investigated.

Linyphone uses an approach which is very similar to the one we chose; however, it differs from our solution in that it is designed to be run completely on the end hosts and requires them to run special software.

We were able to produce a functional application, albeit with several problems. The most severe was the unsatisfying performance: When testing our solution, we learned that the current version of Linyphi does not perform well on the Linksys devices. A new, faster version of Linyphi is under construction, and we believe it may again be extended using the same basic concept of an added SIP proxy and DHT.

Contents

1 Introduction 1

2 Background 2

2.1 Peer-to-Peer (P2P) and mesh technologies 2

2.1.1 Peer-to-Peer (P2P) 2
2.1.2 Mesh network and Mobile Ad-hoc Network (MANET) 4
2.1.3 Distributed Hash Table (DHT) 6
2.1.4 Scalable Source Routing (SSR) 7
2.1.5 Linyphi 12

2.2 OpenWRT and Linksys WRT54GL 16

2.3 Voice over Internet Protocol (VoIP) and Session Initiation Protocol (SIP) 17

2.3.1 Voice over Internet Protocol (VoIP) 17
2.3.2 Session Initiation Protocol (SIP) 17

2.4 Objectives 20

2.5 Existing solutions 21

2.5.1 P2PSIP, SIPDHT and SOSIMPLE 21
2.5.2 Skype 23
2.5.3 Linyphone 24

Our solution: Extending Linyphi 25

3 Design and implementation 27

3.1 Introduction 27

3.2 Global architecture 27

3.3 Storing and retrieving user locations 29

3.3.1 Mirrored storage 29
3.3.2 External DHT 29
3.3.3 Our approach: Internal DHT 31

3.4 Required modifications of existing Linyphi and SSR 32

3.4.1 Packet routing enabled using only approximate SSR IDs 32
3.4.2 Variation of SSR IDs 35

3.5 Implementation of new functionality 39

3.5.1 Motivation 39
3.5.2 Details 40
3.5.3 Hash function 41

3.6 Scenarios 41

3.6.1 A SIP user registers its presence in the network 41
3.6.2 A SIP user invites another SIP user in the network 42
3.6.3 A new router joins the network 44

3.6.4 A router leaves the network 44

3.7 Scope 45

3.7.1 Performance 45
3.7.2 UDP and TCP 45
3.7.3 Internet connectivity 46
3.7.4 Mobility 46
3.7.5 SIP specification compliance 46
3.7.6 Authorization and security 46
3.7.7 Error handling 46

4 Result and evaluation 47

4.1 Introduction 47

4.2 Versions of Linyphi 47

4.3 Encountered problems 48

4.3.1 Lack of IPv6 capable SIP software 48
4.3.2 Linyphi and the Linux kernel 48
4.3.3 No media transfer 49
4.3.4 Linyphi robustness and compatibility 49
4.3.5 No change in SSR ID assignments 50

4.4 Evaluation of functionality 50

4.5 Performance evaluation method 51

4.5.1 Original (abandoned) test plan 51
4.5.2 Revised test plan 54

4.6 Performance results 59

4.7 Performance conclusions 61

4.7.1 Scalability 61
4.7.2 Processing power 61
4.7.3 Our modifications 62

5 Conclusion and future work 63

5.1 Conclusion 63

5.2 Lessons learned 63

5.3 Future work 63

References 65

A Source code 68

B Test data 68

C Useful links 68

List of Figures

Figure 1: Client/server communication (not Peer-to-Peer)..1

Figure 2: Peer-to-Peer communication 3

Figure 3: Mobile Ad-hoc Network (MANET) 1

Figure 4: Client and server in a mesh network..1

Figure 5: SSR’s Chord-like virtual ring..1

Figure 6: SSR routing sample (from [21]) 9

Figure 7: Linyphi physical topology..1

Figure 8: Linyphi packet routing..1

Figure 9: Linksys WRT54GL...16

Figure 10: Sample SIP presence registration 18

Figure 11: Sample SIP invitation..18

Figure 12: P2PSIP/SIPDHT/SOSIMPLE overlay...22

Figure 13: P2PSIP/SIPDHT/SOSIMPLE overlay over SSR 22

Figure 14: Linyphone components (from [19])..24

Figure 15: Linyphi over mesh network..1

Figure 16: Routers forwarding SIP packets differently depending on target domain name..1

Figure 17: A put request to OpenDHT 1

Figure 18: How “default routing” in SSR handles packets to none-existing nodes..............1

Figure 19: How “closest routing” in SSR handles packets to none-existing nodes 1

Figure 20: Illustration of the problem with unevenly distributed SSR IDs 1

Figure 21: Illustration of the problem with unevenly distributed SSR IDs being solved......1

Figure 22: Linyphi extended with the SIP module..1

Figure 23: Sequence diagram of SIP registration using our solution 42

Figure 24: Sequence diagram of SIP registration and invitation using our solution...........43

Figure 25: Physical test setup (abandoned). Numbers under routers indicate SSR IDs. 1

Figure 26: The virtual SSR ring in the abandoned test setup...1

Figure 27: Reference test; unmodified Linyphi running on one Linksys router...................1

Figure 28: Scalability test; unmodified Linyphi running on two Linksys routers 1
Figure 29: Processing power test; Linksys router versus homemade x86 router..................1
Figure 30: Testing our modifications; Original Linyphi versus modified Linyphi...............1

List of tables

Table 1: SSR packet types..11
Table 2: Structure of SSR “Connect” packet 12
Table 3: Linyphi IPv6 address...13
Table 4: MAC-48 of three Linksys WRT54GL routers..35
Table 5: Ten random 48-bit hash values (ordered)...36
Table 6: Distribution of hash value to SSR ID approximation 36
Table 7: Byte wise reversed MAC-48 of three Linksys WRT54GL routers 38
Table 8: Distribution of hash value to reversed SSR ID approximation 38
Table 9: Performance measurements..60

List of abbreviations

AODV	Ad-hoc On-demand Distance Vector (a routing protocol)
AODV-UU	AODV implementation by Uppsala University
DHT	Distributed Hash Table
IEEE	Institute of Electrical and Electronics Engineering
IETF	Internet Engineering Task Force
IP	Internet Protocol
IPv4	Internet Protocol Version 4
IPv6	Internet Protocol Version 6
ITU	International Telecommunication Union
ITU-T	ITU Telecommunication Standardization Sector
MAC, MAC-48	Media Access Control, 48 bits (an address identifying a NIC)
MANET	Mobile Ad-hoc Network
NIC	Network Interface Card
P2P	Peer-to-Peer
QoS	Quality of Service
RFC	Request for Comments
SDP	Session Description Protocol
SIP	Session Initiation Protocol
SSR	Scalable Source Routing
TCP	Transmission Control Protocol
UAC	User Agent Client
UAS	User Agent Server
UDP	User Datagram Protocol
URI	Uniform Resource Identifier
VoIP	Voice over Internet Protocol

1 Introduction

A mesh network is a network in which data is routed through a number of nodes in a way so that any node can communicate with any other node in the network – across several nodes, if necessary (called “multi-hop routing”). A mesh network is usually wireless and designed to handle that nodes are added, disabled or removed to/from it. As a mesh network does not necessarily require any routers, servers or other central structure to function, it can be seen as a peer-to-peer structure. Mesh networks are interesting because they are dynamic, reliable (due to not having a single-point-of-failure), and utilize the network capacity in an optimal way.

Voice over IP (VoIP) telephony has a low cost associated with it compared to traditional telephony, as it uses the Internet infrastructure and computers and removing the need for the parallel system of analogue telephone lines, stations and switches. As the cost of network capacity has been decreasing, VoIP has increased in popularity. It is entirely possible to use most VoIP applications on top of most mesh networks. However, the structured architecture of most VoIP implementations (including SIP, the free and open standard by the IETF) does not leverage the advantages of the unstructured mesh networks.

Our objective is to enable VoIP communication on a mesh network in some way, which leverages the advantages of both technologies. As the existence of servers in a mesh network is a large part of what makes SIP unsuitable for a mesh network, we need to find some way of implementing SIP which does not require servers. The solution should be independent of any remote services, independent of any centralized service, be scalable, and - unlike most existing solutions - require minimal modifications and configurations on the SIP clients.

The rest of this paper is structured as following: In section 2, we further discuss P2P, mesh networks, VoIP, SIP and other relevant technologies to provide the background for our work. In section 3, we describe our solution, all relevant details about it and discuss all design decisions. In section 4, after having implemented the solution we evaluate our solutions functionality and performance. Section 5 provides a conclusion, lessons learned and ideas regarding future work.

2 Background

In section 2.1, we will describe the concept of P2P and its benefits. In section 2.2, we will describe the OpenWRT operating system and the Linksys WRT54GL router, which will play important roles in our work. In section 2.3, we will describe SIP (Session Initiation Protocol) and its benefits, and discuss why it is not suitable to run SIP over P2P networks. In section 2.4 we will establish the objectives of this thesis, and in section 2.5 we will look at some existing solutions and see how they fail to reach these objectives. In section 2.6, we will describe our solution.

2.1 Peer-to-Peer (P2P) and mesh technologies

2.1.1 Peer-to-Peer (P2P)

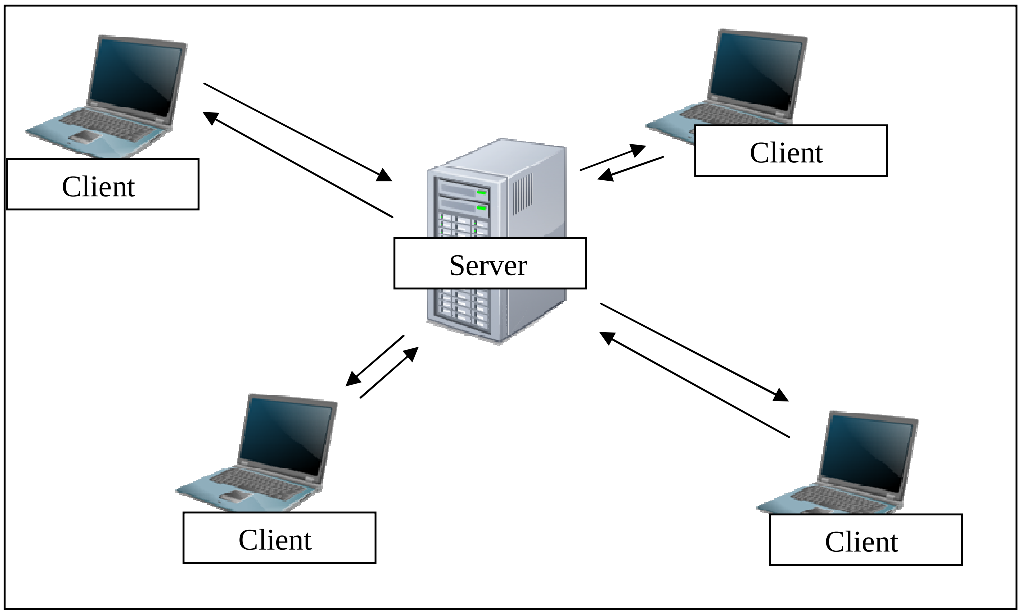


Figure 1: Client/server communication (not Peer-to-Peer)

In most computer communication, a “client-server” structure is used, in which every node acts as either “client” or “server”. For example, a client usually referred to as a web browser connects to a web server to retrieve a web page, or an e-mail client is used to connect to an e-mail server to send and retrieve e-mails. In these and many other services, the use of a client-server structure makes perfect sense. In other cases however, such as instant messaging, VoIP (see section 2.3.1) and file sharing, the need for a server is less obvious – when exchanging information between two computers, it is more efficient to do it directly between the two computers than to pass the information through a third computer.

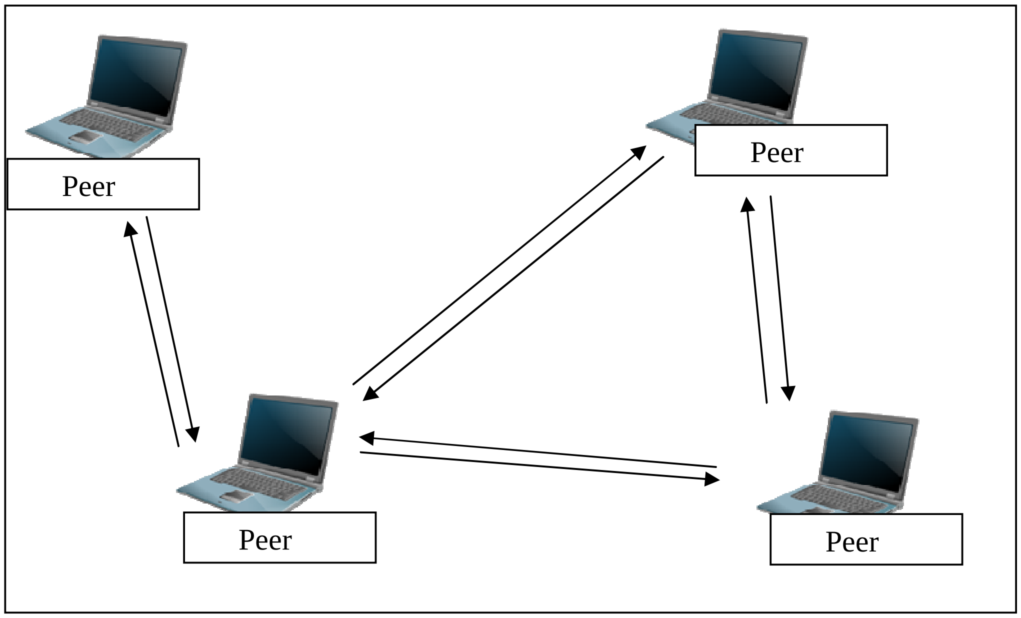


Figure 2: Peer-to-Peer communication

This is where Peer-to-Peer (P2P) networks become useful. As defined in [36], a P2P network is a network where any node is both a resource provider as well as a resource requester. Popular examples of P2P networks are the file sharing networks Gnutella and BitTorrent [1]. The advantage of P2P is that the servers play a smaller role and handle less traffic, requiring less bandwidth, and in some cases can be removed completely. Performance in general is a major advantage of P2P; the success of Gnutella and BitTorrent in particular may be attributed to their scalability and high bandwidth utilization.

A network using a “pure” P2P architecture (one which does not require any servers) is usually self-healing, self-organizing and very scalable. [1]

P2P architectures can be applied on several layers of the network protocol stack. The examples above apply to the application layer, but also nodes in the link layer can be P2P, as described in the next section.

2.1.2 Mesh network and Mobile Ad-hoc Network (MANET)

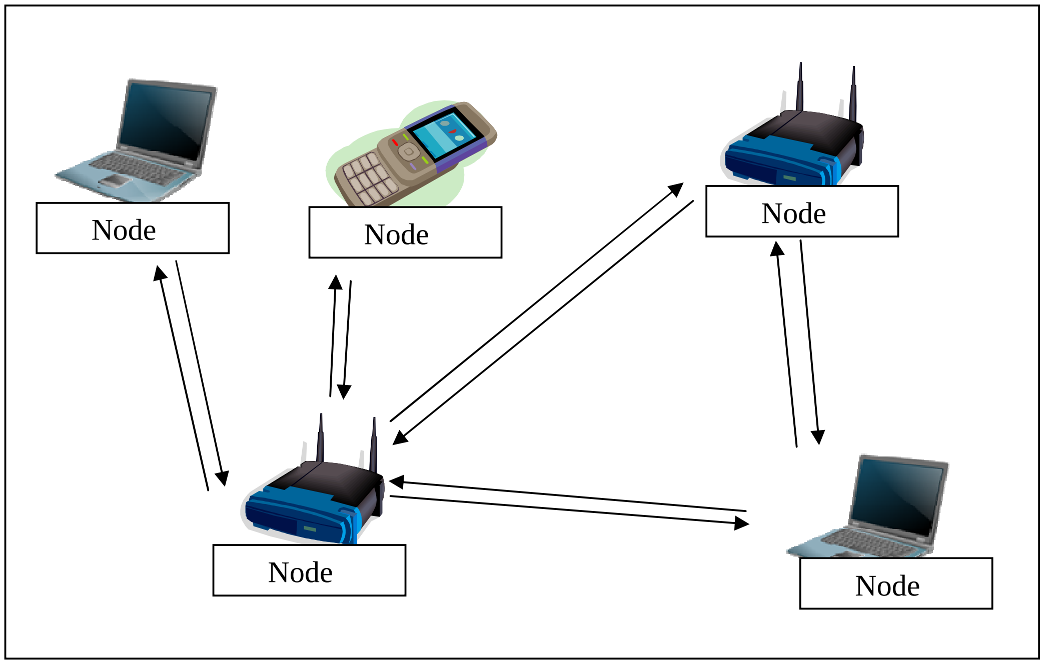


Figure 3: Mobile Ad-hoc Network (MANET)

A *mesh network* is a network in which data is routed through a number of nodes, in a way so that any node can communicate with any other node in the network. If there is no direct connection between two nodes, packets are routed over paths spanning across multiple nodes in the network (called “multi-hop routing”). A mesh network use dynamic routing and is “self-healing”, which means that routing paths are automatically updated when nodes are disabled or are added or removed to/from the network, so that connectivity to all remaining functional nodes are maintained. A mesh network may or may not be wireless, although wireless mesh networks may be the most typical and common application. As a mesh network does not necessarily require routers, servers or other central structure to function, it can be compared to a peer-to-peer structure.

A *mobile ad-hoc network* (MANET) is a type of *wireless* mesh network which also has to handle that the nodes that form the network are highly mobile. The network must quickly adapt as nodes move around, so that routing paths are kept functional and optimal. [27]

There are currently no standard routing protocols for MANETs. Standardization is in progress by the IETF. [26]

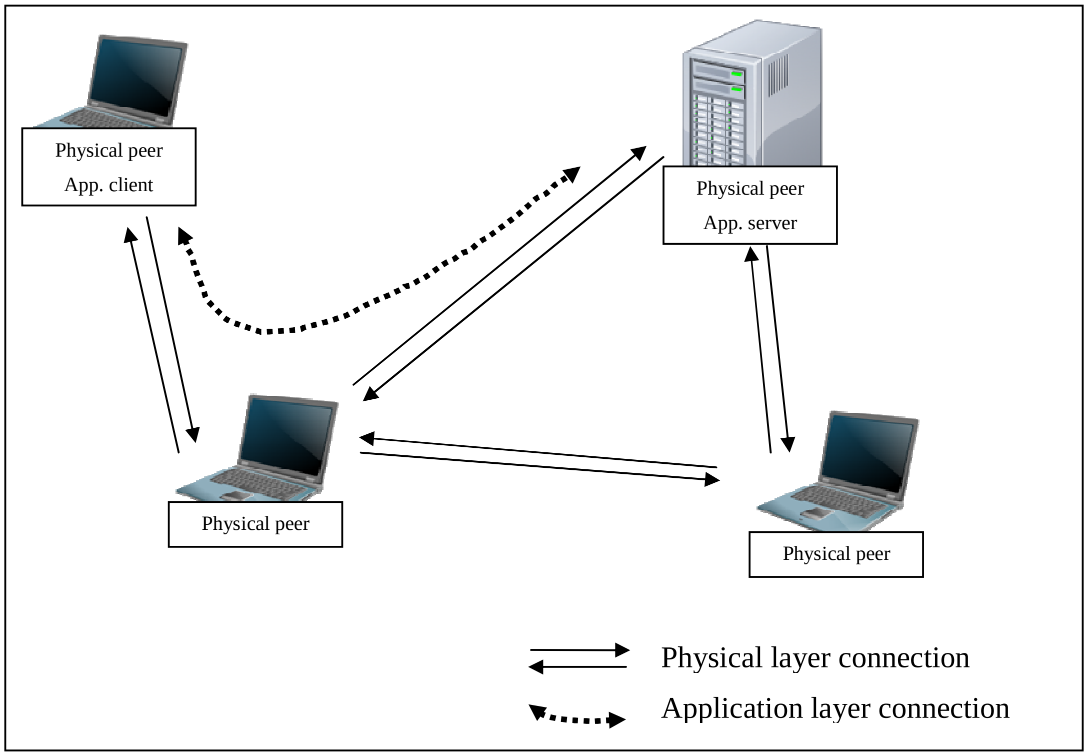


Figure 4: Client and server in a mesh network

Essentially, a mesh network is a P2P structure applied in the bottom layer of the network protocol stack, and has the same advantages as P2P in the application layer. However, as usual with the stack, the type of topology and protocol on the different layers is technically independent of each other. A mesh network can carry regular client-server services throughout the network, while a P2P application can be overlaid over any kind of underlying network structure. However, due to the nature of a mesh network where nodes may join and leave any time, a client-server service, or a “hybrid” P2P service (which requires some kind of server according to the definition in [36]) may be inappropriate since the server may become unreachable.

A mesh network can be configured to be connected to the Internet. The details of this are highly dependent on the implementation. First, one or more of the nodes must be connected to the Internet and will function as a gateway(s). The remaining nodes must know which node(s) that are gateway(s), either by reading local configuration or by receiving or retrieving this from the network.

2.1.3 Distributed Hash Table (DHT)

A Distributed Hash Table (DHT) is a table which contains entries consisting of key-value- (or name-value-) pairs. The table is distributed over a number of connected nodes. Clients of the DHT can request or set a value for any key from any node in the DHT; the request will be forwarded within the DHT to the node which stores or should store the key-value-pair.

Which node should store which entry (or where to find a specific entry) is determined by applying a hash function on the key. A hash function always output some value which is dependent on the input (the same input must always produce the same output) but is well distributed over the value domain. This is the reason for hashing the key: The keys are often not well distributed. If the keys for example consist of user names or any kind of words, some letters will be more common than others and the DHT may not be distributed equally. Different nodes in the DHT are responsible for different intervals of the hash values. How these intervals are decided and how a node is aware of them is determined by the implementation.

A DHT should also be able to take advantage of new nodes that join the network and handle that nodes leave the network if done under controlled circumstances. For example, if we have a very small DHT made up of only two nodes, with a key hash value domain of 0-255: The first node might store all key-value-pairs with key hash values of 0-127, while the second node stores all pairs with key hash values of 128-255. When a third node is added to the DHT, the distribution needs to change so that, for example, the first node stores key-value-pairs with key hash values 0-85, the second 86-171, and the third 172-255. Depending on the implementation, the nodes needs to be updated so that they are aware of this new distribution, and the nodes which no longer are responsible for certain key intervals needs to submit the affected pairs to the new node.

A DHT plays a key role in different types of P2P networks. It is possible to add a *structured overlay* to a P2P network, thus making the inherently unstructured network structured. For example, while the connections in a mesh network are P2P, a structured overlay can be added in the routing protocol to form a structured overlay network. The most common approach to implementing a structured overlay is by implementing a DHT. [41]

2.1.4 Scalable Source Routing (SSR)

Scalable Source Routing (SSR) [21] is a network layer routing approach for mesh networks and MANETs. SSR combines source routing in the physical network layer with a Chord-like virtual ring structure.

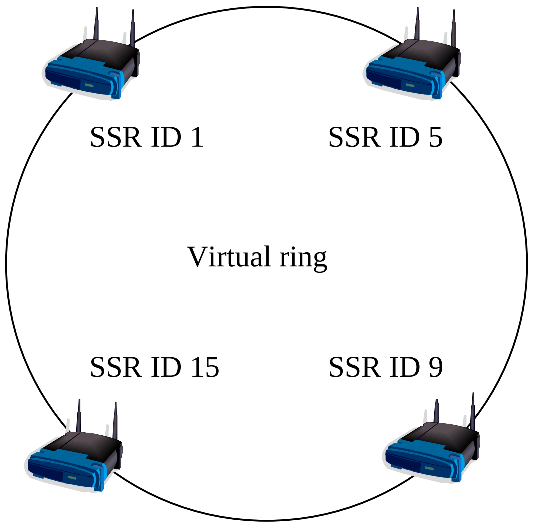
SSR itself does not specify a way to interoperate with Internet; this is left to the implementation.

SSR is presented in more detail in [21].

2.1.4.1 Structure

In SSR, each node is assigned a globally unique address (subsequently referred to as SSR ID). The assignment is left to the implementation, which thus is responsible for somehow supplying a globally unique identifier.

Each node in the network is always aware of the physical path to a node referred to as the *successor*. The successor is the node which SSR ID is the "virtually closest" to the current node, that is, the node with the lowest SSR ID that is higher than the current node's. To the successor, the current node is the predecessor. The address space is looping, thus, the node with the globally lowest SSR ID is the successor to the node with the globally highest. This forms a Chord-like *virtual ring*. A sample with four nodes identified by SSR IDs 1, 5, 9 and 15 is displayed in Figure 5.



Predecessor	**Node**	Successor
15	**1**	5
1	**5**	9
5	**9**	15
9	**15**	1

Figure 5: SSR's Chord-like virtual ring

Note that how the nodes are physically connected does not relate to the ring structure depicted in the figure.

The virtual ring is unidirectional: One nodes virtually closest node is *always its successor*, even if the predecessors SSR ID is numerically closer to the current nodes SSR ID than the successors. Similarly, one nodes virtually most distant node is its predecessor. In a network with only two nodes, both nodes are the predecessor and successor to the other node at the same time.

The routing paths to the successor and predecessor are stored in a routing cache, together with the paths to all nodes physically connected directly to the node, and a number of other nodes which have been cached during different routing procedures explained below. Each node in the network is responsible for handling all packets (or messages) targeted to any target address equal or higher than the nodes own SSR ID, but lower than the successors SSR ID.

2.1.4.2 Operation

Packet forwarding

A packet contains a path known as “source route” containing the route from the source node to a node which is the final destination, or is a node closer to the final destination. While this path has not been traversed, the nodes blindly forward the packets. When the path has been traversed, a new routing decision is made, unless the path was leading all the way to the final destination and thus has been reached, in which case the packet is delivered to the upper layers of the network protocol stack.

When a new routing decision is made, two metrics are compared to calculate which node is the best choice: The physical distance (the number of physical hops between the nodes) and the virtual distance (the numerical difference of the node’s SSR IDs). The node will *always* forward the packet towards a node which is virtually closer to the final destination node than the current node. The path to the selected node is appended to the source route in the SSR packet.

If a node does not know of a node virtually closer to the final destination than itself, and itself is not the final destination, the final destination does not exist (presuming the network is consistent). In this case, the node discards the packet, and sends back a control packet along the source path to notify these nodes that the node is missing.

Figure 6 illustrates a routing procedure for a packet originating from a node with SSR ID 1, to the node with SSR ID 42.

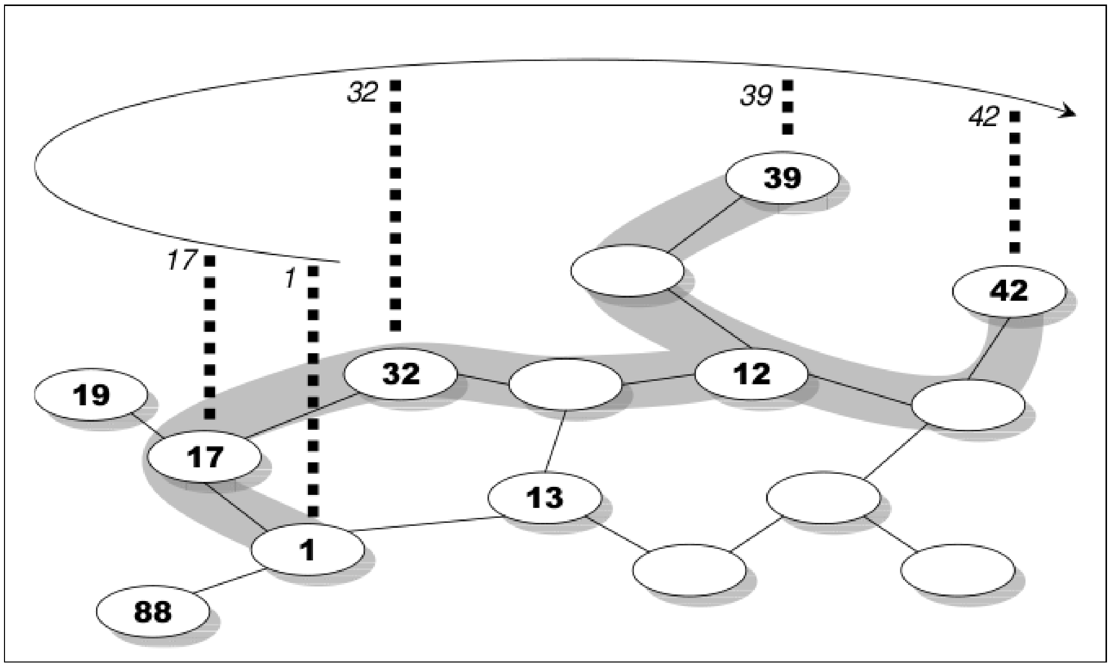


Figure 6: SSR routing sample (from [21])

In this figure, the straight lines indicate the physical connection, the half-circle a portion of the virtual ring, and the shadowed lines the actual path the packet traverse. The routing procedure for this example is as follows:

1. Node with SSR ID 1 ("node 1") receives a packet from an upper layer in the network protocol stack.
2. The node knows the paths to at least node 13, 17 and 88. It forwards the packet to node 17, because of all nodes it is aware of, 17 is the virtually closest to 42.
3. Node 17 knows the paths to at least node 1, 17, 19 and 32. It forwards the packet to node 32, because of all nodes it is aware of, 32 is the virtually closest to 42.
4. Node 32 knows the path to node 39, since it is its successor. It forwards the packet to node 39, because of all nodes it is aware of, 39 is the virtually closest to 42.
5. Node 39 knows the path to node 42, since it is its successor. It forwards the packet to node 42.
6. Node 42 receives the packet and delivers it to the higher layers of the network protocol stack.

Maintaining the virtual ring

This virtual ring must be kept consistent (that is, every node must know the path to the correct successor) and adapt when routers join and leave the network.

To accomplish this, the following procedure is followed when a node A has joined the network:

1. Node A selects the virtually closest node (node B) of the nodes currently in the nodes cache. By design the cache always contains at least all nodes directly physically connected to the current node.
2. Node A sends a *successor notification* message to the selected node.
3. Node B receives the *notification* message. B compares the SSR ID of A, with B's predecessor (node C).
    a. If A = C (they are the same node), B already treats A/C as its predecessor. B sends an *acknowledgement* back to A/C. Node A/C selects B as its successor.
    b. Otherwise, either C is a better successor candidate for A, or A is a better successor candidate for B. An *update* message is sent to both nodes A and C. The update message contains a concatenation of the routes A→B and B → C. B also keeps the source route from the notification message in its cache.
4. Node A and C receives the update messages, and from this message they both learn a path to the other.
    a. If C is a better successor to A (than to B), node A will send a successor notification to C, and the procedure will be repeated from step 1.
    b. If A is a better successor to C (than to B):
        i. If A is the best suitable successor to C that A knows, node A sends a *predecessor notification* message to C.
        ii. If A knows a more suitable successor to C than itself, node A sends a *successor update* message to C.

The result of the process is that all of the nodes involved have mutually agreed on their predecessors and successors. The process spreads and re-iterates until all nodes are mutually correct.

Note that even if this results in a mutually correct network it may not always be globally correct. This problem is solved by letting every node which has an SSR ID larger than its successor (in Figure 5, it is the node with SSR ID 15) broadcast their address.

2.1.4.3 Cache

An SSR packet contains two paths: the actual source route, and a stub route. For each virtual hop, a new piece is added to both paths, but the source path is optimized if any parts of it are redundant. The stub route contains the original path without any optimizations. This is partly because when a node does the optimization, the node is likely to “cut its own branch”, that is, remove the path which lead to the current node. The stub route lets the packet find its way back to the main route. Also, the stub route is needed so that all nodes along the path back to the source can be notified if the destination node has become unavailable so that they can remove the cached paths to it or find an alternative route.

SSR does not require any control traffic when regular traffic is absent. Since the cache may timeout, it will have to be rebuilt if the network has been idle for an extended period of time.

2.1.4.4 Packet format

SSR utilize several different types of packets. The first byte of an SSR packet identifies the packet type, as described in Table 1.

Value of first byte	**Packet type**
`0x01`	Random Walk
`0x03`	Connect (regular data packet)
`0x04`	New Node Announce
`0x05`	Scout Lost Neighbor [sic]
`0x06`	Unreachable
`0x07`	Neighbor [sic] Notification
`0x08`	Neighbor [sic] Update
`0x09`	Link Broken
`0x0A`	Path Announce
`0x0B`	Max Node Announce

Table 1: SSR packet types

The content of the rest of the packets differ between the different packet types. The “Connect” packet type, which is used for regular data packets, is structured as indicated in Table 2.

Number of bytes (N = length of SSR ID)	**Content**
1	Packet type, `0x03`
2	Length of stub route
N*(length of stub route)	Stub route
2	Length of source route
N*(length of source route)	Source route
1	Hop count (how far the source route has been pursued)
N	Current destination
N	Neighbor [sic] destination
N	Final (original) destination
4	Payload type (Integer, 4 = IPv4, 6 = IPv6, 0 = None, 10 = Deliver to application layer)
Rest of packet	Payload (for example, a full IPv6 packet including IPv6 headers)

Table 2: Structure of SSR “Connect” packet

2.1.4.5 Evaluation

SSR has been evaluated in simulations and it has been found to be more efficient and more reliable than AODV and DSR. It has been proved to be scalable well beyond 10,000 nodes, and with low mobility and low node churn, and to have delivery rates of more than 90% in networks with up to 125,000 nodes.

2.1.5 Linyphi

Linyphi is an implementation of the SSR protocol discussed in the previous section, 2.1.4. Between the routers in an SSR network, SSR is used. However, the routing is completely transparent to the hosts, as Linyphi only uses IPv6 when communicating with them (see Figure 7 below). This allows SSR to be used without any modifications to the end hosts, except enabling IPv6 if not done by default. Note that this limits the use of Linyphi to IPv6 enabled applications.

Since the hosts in a Linyphi network are unaware of SSR, only the routers are SSR nodes and have SSR IDs. The problem of assigning globally unique SSR IDs for each node are solved by simply using the MAC-48 addresses of the routers, which by design are always globally unique.

Both hosts and routers in a Linyphi network can also be identified by an IPv6 address, in the following way:

Byte	0	1 6	7	8 15
Example	`01`	`00 1B FC 8D 0C AC`	`02`	`00 19 CB FF FE 06 BB F8`
Description	SSR prefix	SSR ID, MAC-48 of router	If.	MAC-48 of end host

Table 3: Linyphi IPv6 address

- **SSR Prefix:** Always `01`, a prefix that identifies that the address is assigned by Linyphi.
- **SSR ID, MAC-48 of router:** The SSR ID, which equals the MAC-48 address of the router to which the end host is attached. If the IPv6 address identifies a router, this is the MAC-48 of the router.
- **If. (Interface):** A number identifying the interface on the router, to which the end host is connected. If the IPv6 address identifies a router, any number can be used as long as there is a corresponding interface on the router.
- **MAC-48 of end host:** The MAC-48 address of the end host, converted to 64-bit according to the conventions described in [40] (adding `FF FE` to the middle of the address). If the IPv6 address is that of a router itself, the same MAC-48 is used in both bytes 1-6 and 8-15 (in the latter converted to 64-bit).

As the router assigns the IPv6 addresses for each end host in what to the clients appear as regular IPv6 address assignment mechanisms, the end hosts does not need to know to which interface they are connected to, or which MAC-48 address the router has.

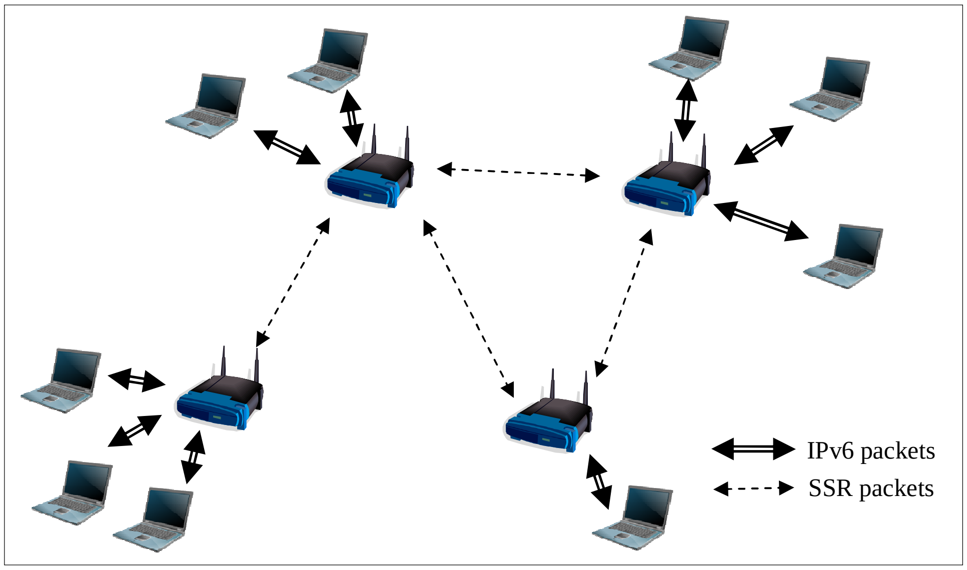


Figure 7: Linyphi physical topology

When an end host or a non-Linyphi router sends a packet to an SSR router, it is always sent using standard IPv6 (as previously mentioned). Linyphi extracts bytes 1-6 from the IPv6 address to determine the SSR ID of the packet's destination SSR node. If that node is the router itself, it is passed on as IPv6 to the appropriate interface (as seen in byte 7 in the IPv6 address) on the router.

If the SSR ID embedded in the IPv6 packet is *not* the current node, an SSR header (see section 2.1.4.4) is pre-pended to the IPv6 packet. SSR is then used as in section 2.1.4.2 to route the packet closer to the destination node.

When a Linyphi router receives an SSR packet, it either forwards it to a node closer to the final destination (as described in section 2.1.4.2), or if the router is the final destination node of the SSR, removes the SSR header. It then reads the interface byte of the IPv6 address (byte 7) to determine which of the router's interfaces to forward the packet on, and forwards it to this.

Each interface (commonly, one WAN, one LAN and one WLAN) on a router is essentially a switched hub (often simply referred to as "switch"), hence the need to include also the MAC-48 of the end host in the IPv6 address.

These procedures are summarized in Figure 8.

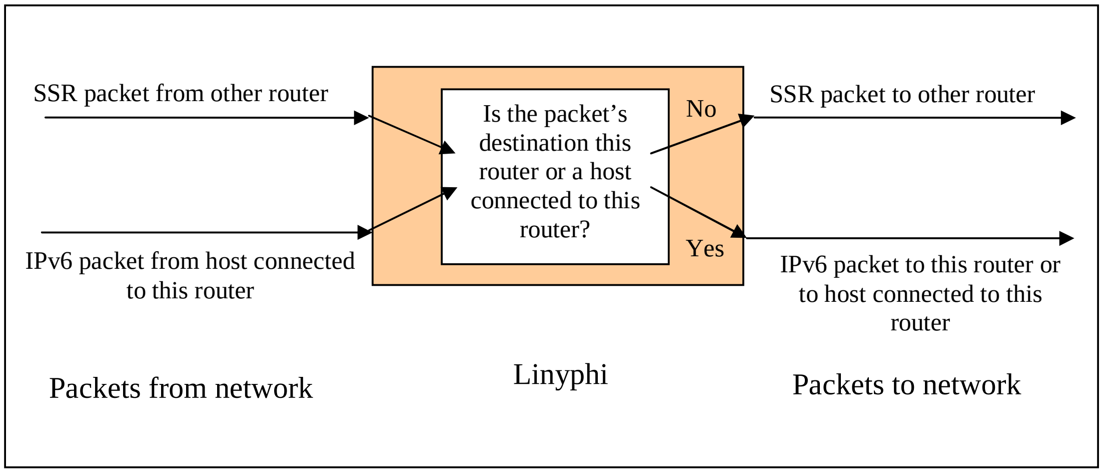

SSR packet from other router
IPv6 packet from host connected to this router
Is the packet's destination this router or a host connected to this router?
No
Yes
SSR packet to other router
IPv6 packet to this router or to host connected to this router
Packets from network
Linyphi
Packets to network

Figure 8: Linyphi packet routing

No different than in IPv4, the end hosts are not automatically made aware of other hosts connected to neither the same router as itself, or to any other router. Other mechanisms outside of Linyphi and IPv6 must be used to learn addresses of other hosts in the network.

A Linyphi network can be connected to the IPv4 Internet. Off-network packets are directed towards a designated gateway in the network. Linyphi's configuration file defines which router that is. Then, this router/gateway performs name address translation (NAT) to translate the IPv6 address of the packet into an IPv4 address, and the reverse for incoming packets.

Linyphi is implemented as software for the Linux operating system, which makes it possible to run it on most hardware capable of running Linux.

A library exists, LibIgor, which adds a DHT API, implementing DHT functionality which leverages the properties of SSR. However, documentation of this API is minimal. [20]

More information about Linyphi is available in [2] and on its website at [42].

2.2 OpenWRT and Linksys WRT54GL

Figure 9: Linksys WRT54GL

Linksys WRT54GL is a network router with a built in wireless 54 Mbps 802.11b/g Ethernet access point and a 4-port 10/100 Mbps switch[1].

The firmware in WRT54GL is based on Linux and thus its source code is publically available as OpenSource. This has enabled the development of a number of third-party firmwares, such as OpenWRT [5], DD-WRT [37] and Tomato [38], among others. These firmwares target different types of users; OpenWRT is primarily aimed towards advanced users and allows vast opportunities for customization.

From the perspective of our work, the most important possibilities WRT54GL and OpenWRT provides are that of running custom routing protocols (most importantly, SSR by running Linyphi) and creating different testing environments.

Note that OpenWRT is not limited to the Linksys WRT54GL; several other routers of various brands are capable of running the firmware.

[1] Actually, the WRT54GL is essentially one Wireless Ethernet NIC and one Ethernet NIC. The latter is hardwired to a programmable 6-port Ethernet switch. The remaining five ports are accessible from the back of the router, and are labeled as 1-4 and "WAN". The actual distinction and the routing between these are made entirely through software.

2.3 Voice over Internet Protocol (VoIP) and Session Initiation Protocol (SIP)

2.3.1 Voice over Internet Protocol (VoIP)

Voice over Internet Protocol (VoIP) is a term describing the transfer of digitized voice over the Internet using the Internet Protocol (IP), in effect, making phone calls using the internet infrastructure rather than the traditional telephone network.

The main advantage of VoIP is the low cost associated with it compared to traditional phone calls, by using the Internet infrastructure and computers and removing the need for the parallel system of analogue telephone lines, stations and switches. [29]

The drawbacks are the lack of Quality of Service (QoS). In a traditional telephone network, a “line” is reserved at the establishment of a phone call, ensuring the quality of the phone calls is stabile under the full duration of the call. On the Internet, no Quality of Service is widely implemented. VoIP calls over a busy or unreliable route may therefore suffer from varying sound quality and delay, or even disconnection or the inability to make phone calls. [28]

2.3.2 Session Initiation Protocol (SIP)

Session Initiation Protocol (SIP), defined in [17] is an application layer protocol, which provides a way of establishing and maintaining communication sessions between hosts, often for audio (VoIP, see section 2.3.1), video and instant messaging. The protocol also specifies proxy server functionality, which provides authentication, routing, enforcement of policies, and locating facilities.

SIP does *not* specify the media or compression to use during the actual communication. Media capabilities of hosts are attached to SIP messages as payload, and are described using the Session Description Protocol (SDP) [30].

The advantage of SIP is that it provides a standardized and open way to initiate sessions of any kind, not only currently existing VoIP or instant messaging, but for use with any software or media transfer protocol.

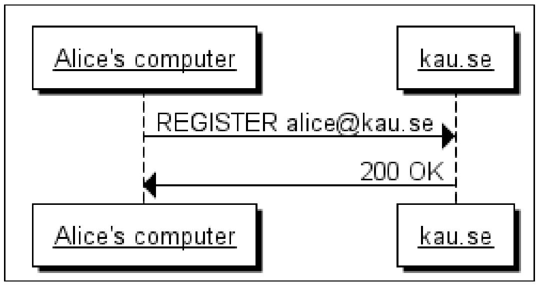


Figure 10: Sample SIP presence registration

Similar to e-mail users, each SIP user has a unique identifier called SIP URI [31], such as "`sip:alice@kau.se`". In this SIP URI, "`alice`" is the user name of the SIP user, and "`kau.se`" is the SIP provider. The `kau.se` host will serve as a proxy for Alice; that is, when Alice connects to some network containing a `kau.se` host, her UAC (User Agent Client, or SIP client) registers her presence on the SIP proxy located at "`kau.se`" by sending it a `REGISTER` request. This is illustrated in Figure 10. (Similarly to the HTTP protocol, the result code "2xx" indicate success, "3xx" specify a redirection, "4xx" indicate unauthorized access, and "5xx" indicate that an error has occurred on the server.)

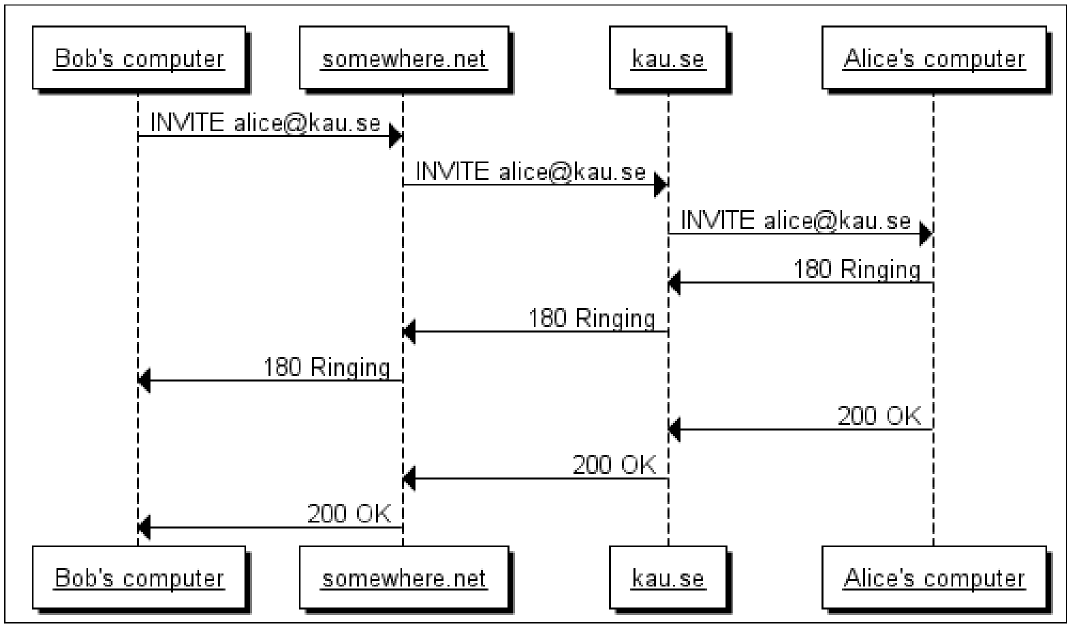


Figure 11: Sample SIP invitation

When another SIP user such as "`bob@somewhere.net`" wishes to call Alice, he will enter Alice's SIP URI which he/she must know somehow. As illustrated in Figure 11 (assuming Bob has already registered his presence) Bob's UAC will send an `INVITE` request to Bob's proxy server, `somewhere.net`. This proxy server will then forward the request to the proxy server at `kau.se` (as extracted from the SIP URI), which will then forward the request to Alice's UAC, causing her SIP software (or hardware) to start ringing. This is indicated by a "180 Ringing" reply. When Alice answers the call, a 200 OK will be replied to Bob and the session has been initiated. Media exchange is then sent directly between Bob's and Alice's computer, without going through the proxy servers.

A proxy server may be either stateful or stateless. As opposed to a stateful, a stateless proxy does only forward packets up or downstream, without maintaining the state of ongoing session initiation transactions.

A registrar only handles `REGISTER` requests to store the locations and availability of SIP user's locations.

There is an extension to SIP named SIMPLE which adapts it further for use for instant messaging (IM).

Any mesh network which can run arbitrary IP traffic can run SIP. This includes Linyphi, although it is limited to IPv6. If the mesh network can be connected to the Internet, which also includes Linyphi, there are a several ways users can communicate over the Internet and within a mesh network using SIP. However, there are several drawbacks of running SIP in a mesh network. We attempt to explain those below in a number of scenarios, which are all assuming all users are connected to same SIP server for simplicity's sake.

1) A SIP user is on a mesh network. He/she is using a SIP server on the Internet.
    a. He/she communicates with a SIP user on the same mesh network.
    b. He/she communicates with a SIP user on the Internet, or in another mesh network also connected to the Internet.
2) A SIP user is on a mesh network. He/she is using a SIP server on the mesh network.
    a. He/she communicates with a SIP user on the same mesh network.

b. He/she communicates with a SIP user on the Internet, or in another mesh network also connected to the Internet.

All scenarios should work without any problems, but none of them are optimal or take advantage of the properties of a mesh network, as explained below.

Scenario 1a and 1b: A SIP server located in the mesh network (as in Scenario 2) would be closer and be more accessible to the SIP clients in the mesh network. This is especially true for scenario 1a: If the SIP server would be located in the same mesh network as both clients, the SIP service would be completely independent of the Internet, and likely keep the number of hops minimal without the need for SIP requests to traverse the Internet.

Scenario 2a and 2b: Running a client-server structured application such as SIP over the P2P structure of a mesh network is not an optimal configuration: It works, but does not take advantage of the properties of the mesh network, such as its scalability and reliability. A node acting as SIP server may become available, thus making the entire SIP service in the network unavailable.

Scenario 1b and 2b: The performance of these scenarios depends on how the SIP server is located in relation to the both SIP users (or their mesh networks). Packets have to traverse the Internet in either scenario. The performance of 1b depends on how close the SIP server is to any of the two mesh networks or users. Moving a SIP server from the Internet as in 1b into one of the mesh networks as in 2b, would not necessarily – but could – result in better performance than just moving the SIP server closer to one of the mesh networks, depending for example on the size and performance of the mesh network. However, it would certainly introduce the other drawbacks discussed above.

In short: while SIP and mesh networks are compatible, it is not very optimal to keep SIP servers inside a mesh network due to the P2P nature of mesh networks, neither is using a SIP server outside of a mesh network when several SIP users are connected in one single SIP network due to the unnecessary “Internet detour”.

2.4 Objectives

In previous sections we described several technologies relevant to this thesis, we described the advantages of mesh networks (dynamic networks comprised of mobile nodes), and SIP (a standardized way to initiate sessions of any kind). We also mentioned that the requirement of servers for SIP functionality makes it unsuitable for mesh networks (such as networks running Linyphi).

Our objective is therefore to enable SIP communication on a mesh network in some way, which leverages the advantages of both technologies. As the existence of servers in a mesh network is a large part of what makes SIP unsuitable for a mesh network, we need to find some way of implementing SIP which does not require servers.

The solution should be independent of any remote (such as Internet) services, independent of any centralized service within the network, be scalable, and require minimal configurations or modifications on the SIP clients.

2.5 Existing solutions

In this section, we will describe currently existing technologies which allows serverless SIP, and their advantages and disadvantages.

2.5.1 P2PSIP, SIPDHT and SOSIMPLE

The IETF P2PSIP work group [44] has assembled a number of projects, including the ones listed below.

In all of the projects, the centralized SIP servers are replaced by DHTs or similar technologies. The DHT is then used for locating other users, while the SIP protocol is used for the remaining communication. This forms a P2P overlay in the application layer. Thus, it can be used on top of any network protocol, including a regular TCP/IP star networks or an SSR network. This is illustrated in Figure 12. This overlay adds overhead, and the overall efficiency of the network depends on the underlying layer.

Using an SSR network (or any other P2P/mesh network) as underlying network is not a feasible solution. SSR in particular is itself a P2P overlay over the physical layer. Although SSR considers physical distance in its routing mechanisms, having another overlay adds significant overhead and would bring the actual paths packets traverse even further from the shortest path. This is illustrated in Figure 13.

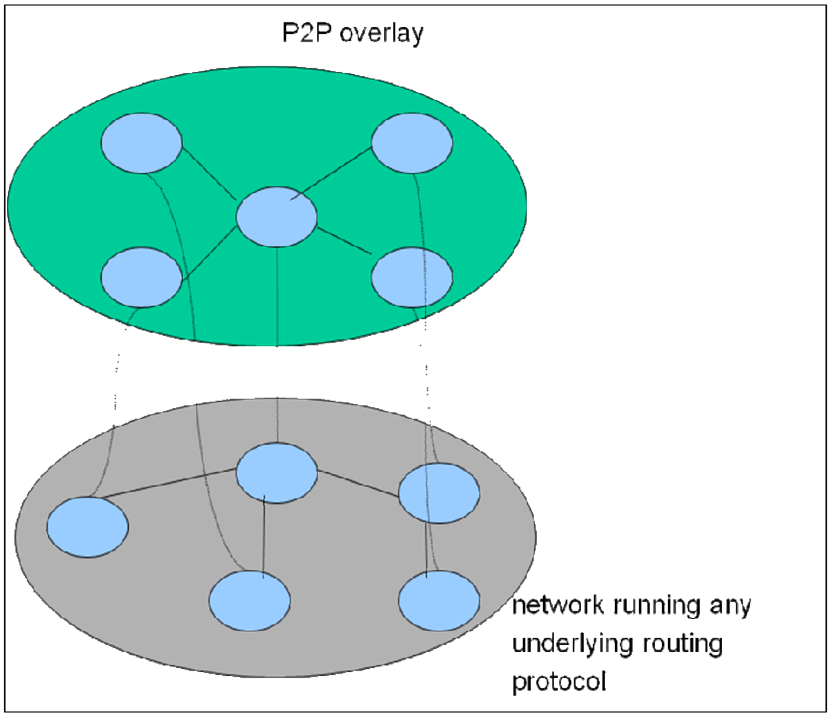


Figure 12: P2PSIP/SIPDHT/SOSIMPLE overlay

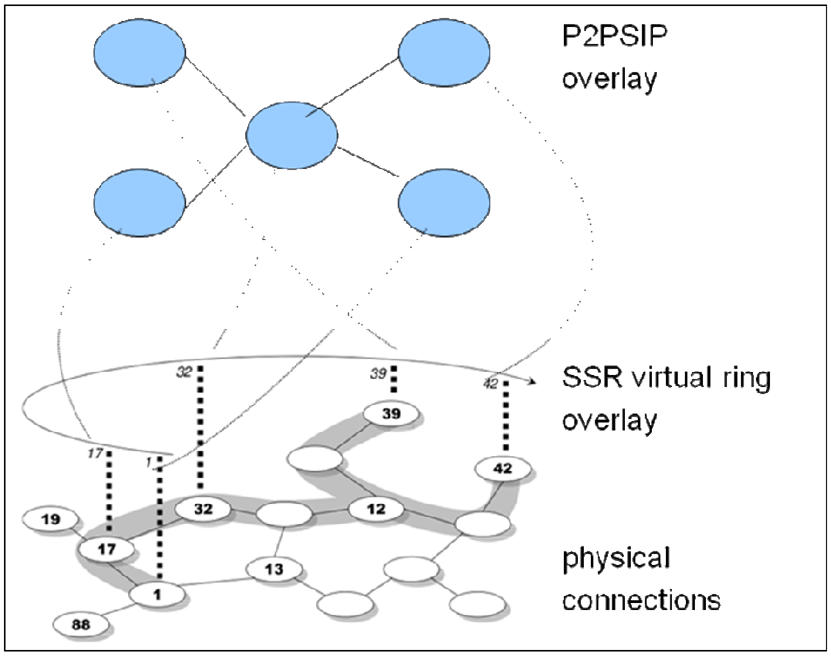


Figure 13: P2PSIP/SIPDHT/SOSIMPLE overlay over SSR

2.5.1.1 P2PSIP by University of Columbia

In this projects approach, upon a client registration, a specific SIP client both tries to find the SIP server using DNS (as any other SIP client would), but is also extended so that it also tries to find and join a Chord P2P overlay using one of several different methods, such as multicasting or an external service.

Thus, a SIP client can both be connected to a regular SIP server, and simultaneously form a P2P overlay. Connections with other P2PSIP clients are managed in the overlay, while connections with standard SIP clients are managed by the regular SIP server. Standard SIP clients will not be able to leverage the P2P overlay.

For more details, see [33].

2.5.1.2 SOSIMPLE by College of William and Mary

In this projects approach, a Chord-based DHT is implemented and the traffic required is added as additional information in the SIP packets. Very few details exist regarding the implementation; the latest publication regarding this project is from 2005 and makes no statement about compatibility with standard SIP clients. For more details, see [32].

2.5.1.3 SIPDHT by Nokia

In this projects approach, a Passive Content Addressable Network (PCAN) is used instead of a DHT. It differs substantially to a DHT in that a new node can only be added to the network if invited by an existing node. The gives improved control over which nodes constitute the network, but a PCAN scales poorly compared to a DHT.

The clients constitute the network and the PCAN functionality is added to the SIP clients. A standard SIP client can connect to the network by configuring any of the SIPDHT nodes as a server, but will not carry any part of the PCAN; it will thus function as a SIP client to one of the nodes in the network which will act as a SIP server.

For more details, see [34].

2.5.2 Skype

Skype uses P2P to maximize performance by sending information through other Skype clients. However, Skype requires a connection to a central authorization service. Also, it uses a closed proprietary protocol, meaning there is no official documentation on this protocol, which makes it hard and possibly even illegal to implement. Some reverse-engineering has revealed that Skype uses a method of flooding the local network to find other nearby hosts, which constitutes wasteful usage of network capacity. [35]

2.5.3 Linyphone

[19] describes an implementation of a solution where the user can join a mesh network arbitrarily using a standard SIP client and a slightly modified version of Linyphi, all running on the users local device.

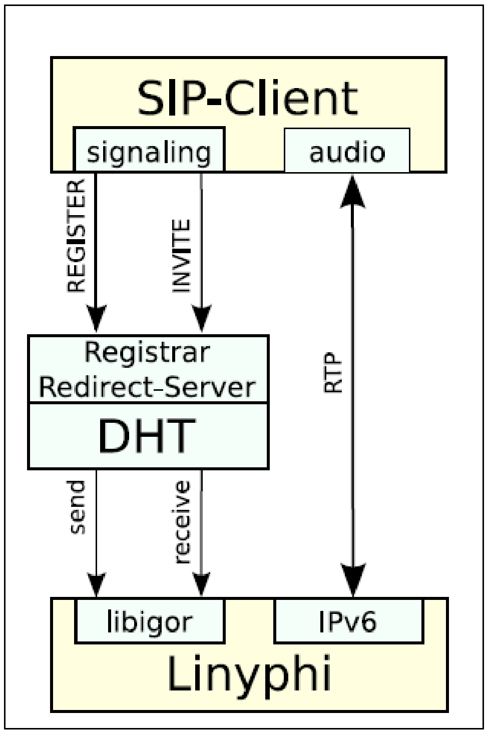


Figure 14: Linyphone components (from [19])

The solution consists of three software components:

- SIP Client: Any standard SIP client, configured to use a SIP server running at the local device as a proxy server.
- SIP Registrar/Redirect-Server: A custom SIP proxy/registrar based on the SofiaSip library, but is extended to control a DHT through the LibIgor API [20].
- Linyphi: Linyphi extended with the LibIgor API.

This system is designed to be contained completely on one single device. Also, documentation of both the entire solution and the LibIgor API is minimal.

Unlike P2PSIP/SIPDHT/SOSIMPLE, no P2P overlay is added; instead the mechanisms of SSR are directly leveraged. While SSR is itself an overlay, it uses a routing approach which considers both physical and virtual distance, reducing the negative effects of the added overlay.

Our solution: Extending Linyphi

As discussed above, neither the P2PSIP concept nor Skype fulfil our requirements of not requiring any modifications on the client. However, our solution will be very similar to the Linyphone approach. This will be further discussed below.

We will extend Linyphi – which as previously mentioned is an implementation of SSR – by adding our own DHT functionality and a simple SIP proxy to the routers. Different from the Linyphone approach, this will all run on every router in the mesh network. Then, any user with a standard SIP client can join the network and become fully available to all other SIP users in the network, only by configuring its client to use any router using the modified Linyphi as SIP proxy.

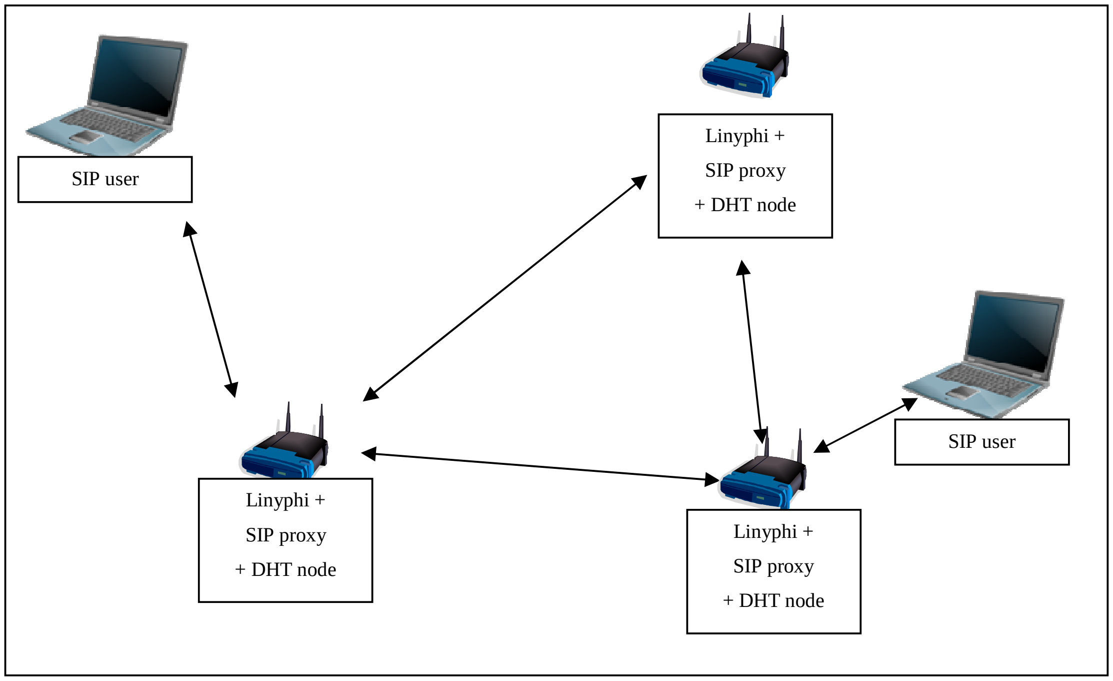


Figure 15: Linyphi over mesh network

The major differences compared to the Linyphone solution, are that Linyphi is ran on routers, and that Linyphi will be extended with SIP and DHT components. On the clients, only a standard SIP client needs to be used. In Linyphone, all components are placed on the end host device.

All details and their motivations will be given in section 3.

3 Design and implementation

3.1 Introduction

In this section, we will describe the details of the design and implementation of our solution.

Essentially, our solution consists of two major design decisions:

- Global architecture: How to allow a user to register its presence in the network and to connect to other SIP users
- Storing and retrieving current user locations: How, and on which node(s) in the network, to store the registered user's presence and exact location (a binding between the user's SIP URI and current IP address of the computer where the SIP client is being run).

These design decisions are explained in section 3.2 and 3.3 respectively. We will then describe what changes needs to be done to the existing solutions to make them compatible with our new functionality in section 3.4, followed by a description how the new functionality will be added in 3.5. In 3.6 we will describe some typical scenarios, and in 3.7 we will describe the scope of the project, explaining which functionality we will and will not implement.

3.2 Global architecture

The SIP user that joins the network must somehow announce its current location to the rest of the network, so that other SIP users can connect to it. It must also be possible for a user to connect to any other SIP user without him/her knowing the other SIP user's location. In a normal SIP system, this is handled by the SIP proxies and registrars – the servers in a SIP network. However, our system must work without any such servers. Therefore, the server functionality will be implemented in the routers, in such way that all routers in the network together act towards the clients as one single standard stateless SIP proxy, equally accessible on all of the routers IP addresses. (In section 2.3.2 we described how the most essential communication is done with standard SIP; these diagrams apply also to this architecture, except all routers can together be seen as one single SIP proxy server, to which all clients are connected). This way, all functionality for registering and inviting other users are already implemented in the SIP clients and therefore requires no modifications, except setting the IPv6 address of the local router as SIP proxy in the SIP client's configuration. It should be possible to modify a SIP client to do this automatically. As described in section 2.3.2, a stateless proxy only forward packets, without maintaining the state of any sessions.

When a router receives a SIP request, it parses the request to determine the domain name of the SIP target server (for example, "`kau.se`" in "`alice@kau.se`"). If the domain name is a specially defined one, such as "`localmesh`", the SIP request is treated within the network using our implementation, and if not, our custom SIP packet handling immediately gives the packet back to the regular routing mechanisms, allowing the package to continue towards the "normal" SIP proxy (for example, the one associated with "`kau.se`"). Note however that this functionality will only partially be implemented: Our implementation will assume all SIP messages directed to a router are meant to be treated within the local Linyphi SSR network, and no special domain is defined. In a future development, this special domain could be defined for example in the Linyphi configuration file on each router.

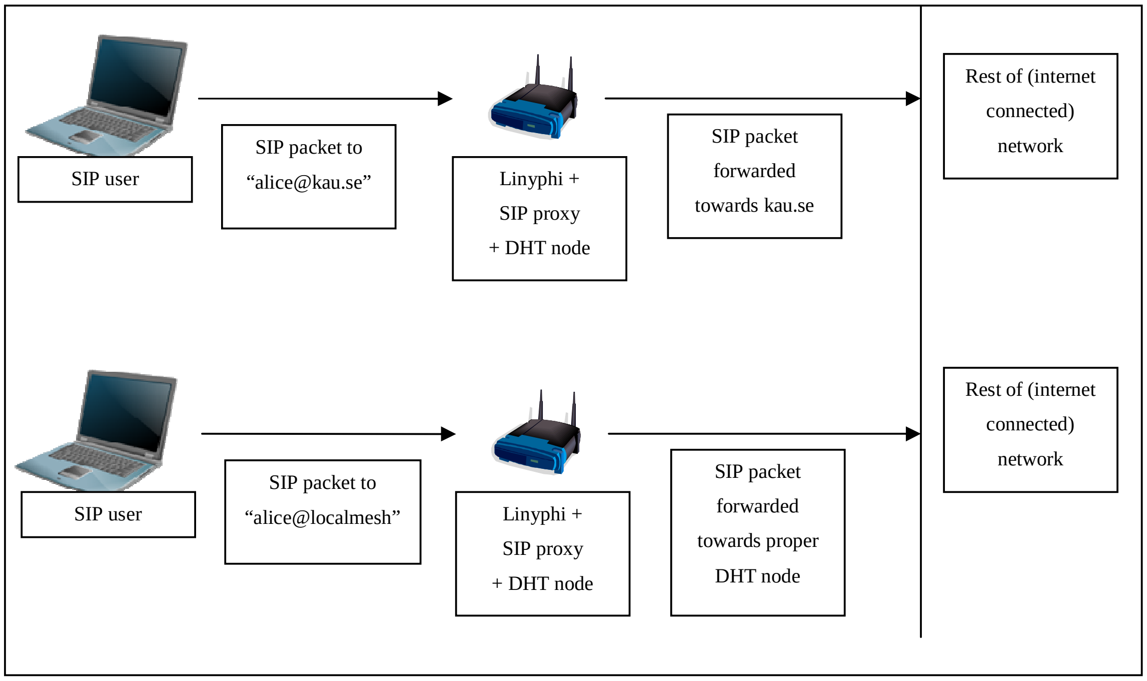

SIP user
SIP packet to "alice@kau.se"
Linyphi + SIP proxy + DHT node
SIP packet forwarded towards kau.se
Rest of (internet connected) network
SIP user
SIP packet to "alice@localmesh"
Linyphi + SIP proxy + DHT node
SIP packet forwarded towards proper DHT node
Rest of (internet connected) network

Figure 16: Routers forwarding SIP packets differently depending on target domain name

3.3 Storing and retrieving user locations

In the previous section, we described that the clients will use standard SIP and will connect to local router to register themselves and to make a phone calls to anyone who has registered in the same network. In this section, we will describe and discuss different methods of storing the different users' locations in the network.

3.3.1 Mirrored storage

Every user's location is stored on each router in the network. This makes finding a user's location very efficient, but was ruled out since such a storage method does not scale well. When a user connects or leaves the network, the location of this user must be added or removed to/from each router in the network, using some flooding mechanism. If the number of routers is $n_{routers}$ and the number of registered SIP users is n_{users}, the total number of entries in the network would have to be:

$$n_{entries} = (n_{users})^{n_{routers}}$$

This is a $O(n^n)$ expression which scales very poorly.

3.3.2 External DHT

With this approach an external DHT service is used, such as OpenDHT [25]. OpenDHT is a publically (over the Internet) accessible DHT running on approximately 200-300 nodes on PlanetLab, which allows users to store arbitrary information. The service is implemented externally, that is, the nodes are spread across the Internet.

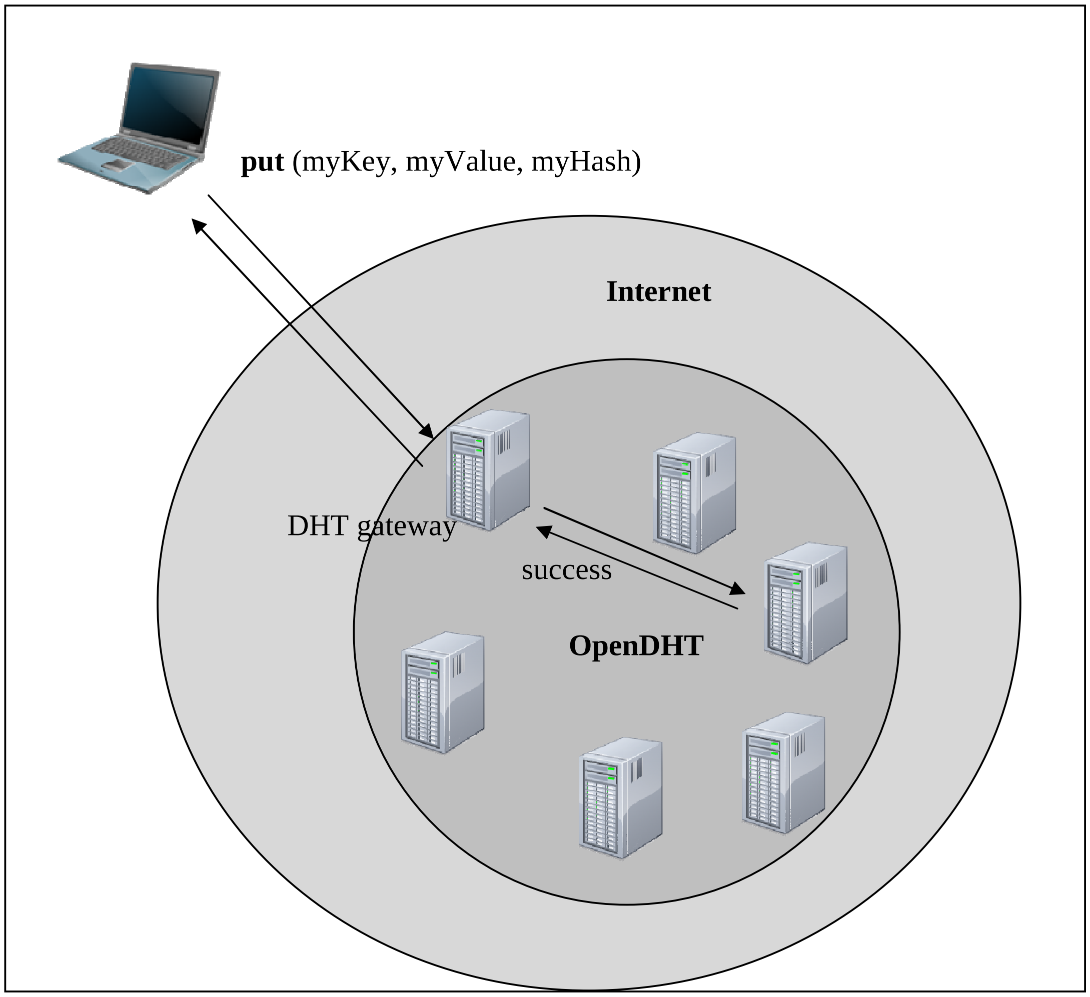


Figure 17: A put request to OpenDHT

Figure 17 shows how a “put” request of a key-value-secret_hash-triple that is sent to one arbitrary node in OpenDHT. The node (which subsequently is referred to as the gateway) forwards the request to the node which is currently responsible for keys such as “myKey”, which depends on the current topology, number of nodes, etc. This node then stores the triple, and a result response is sent back to the requesting client. As several applications share this service, problems are likely to occur if two applications store a value using the same key. That is the reason for the secret_hash being used; it is used as a second, application-specific key.

This service could be used to store SIP presence information. This approach would scale better (only one entry would be required for every user), would be very easy to implement since the DHT facility already exist, and would circumvent problems such as the mobility of nodes. However, it would make the SIP functionality in the network dependent on the external DHT service and on an Internet connection, even to reach SIP users located in the same building.

3.3.3 Our approach: Internal DHT

The approach we have chosen is to implement a DHT on the routers themselves. An internal DHT will scale better than the mirrored storage; each user presence/location is only stored on one router/node. Thus the total number of entries in the network would equal the number of SIP users. Also, the network will not be dependent on an external service or an Internet connection. The disadvantage is the added complexity and the additional work of creating or configuring a DHT implementation on the routers.

There will be one entry in the DHT per SIP user, containing its SIP URI and its current location (its IPv6 address). The value to hash (the DHT key) will be the SIP URI. Each entry will be stored on the router whose SSR ID is closest to the hashed key. This is achieved fairly easy using the functionality of the SSR routing: Normally, in every hop, each routed packet is forwarded to the router whose SSR ID is closest to the destination SSR ID (that is, virtually closer to the destination). If a router does not know of another router virtually closer to the destination than itself - and is not itself the destination - the packet will be discarded. If we change this as described in section 3.4.1 so that some special type of packets are delivered to the current router if no other virtually closer router exist, we can send packets to arbitrary SSR IDs derived from the hashed keys, and be sure they arrive at the router with the closest matching SSR ID in the network.

One problem with this is that while the keys (hashes of SIP URIs) will be spread out evenly over the hash value domain, the SSR IDs will not be. As an example, the SSR IDs (and MAC addresses) of our Linksys routers in our test bed only differs in the 16 least significant bits. This is a problem, because the virtual SSR IDs will be relatively similar, thus the DHT will not be spread equally over the nodes. The problem and the solution are explained more thoroughly in section 3.4.2.

For simplicity, communication between the nodes will be done using SIP as well. Actually, when any router receives a SIP request (such as `REGISTER` or `INVITE`) it will simply forward it with minimal changes towards that router which is responsible for handling the request. For examples of how packets will be sent nodes clients in different scenarios, see section 3.6.

3.4 Required modifications of existing Linyphi and SSR

3.4.1 Packet routing enabled using only approximate SSR IDs

As mentioned in section 3.3.3, to implement the DHT functionality the routers needs to be able to forward any SIP request to the router which has the closest SSR ID to the hash value of the SIP URI of the forwarded packet.

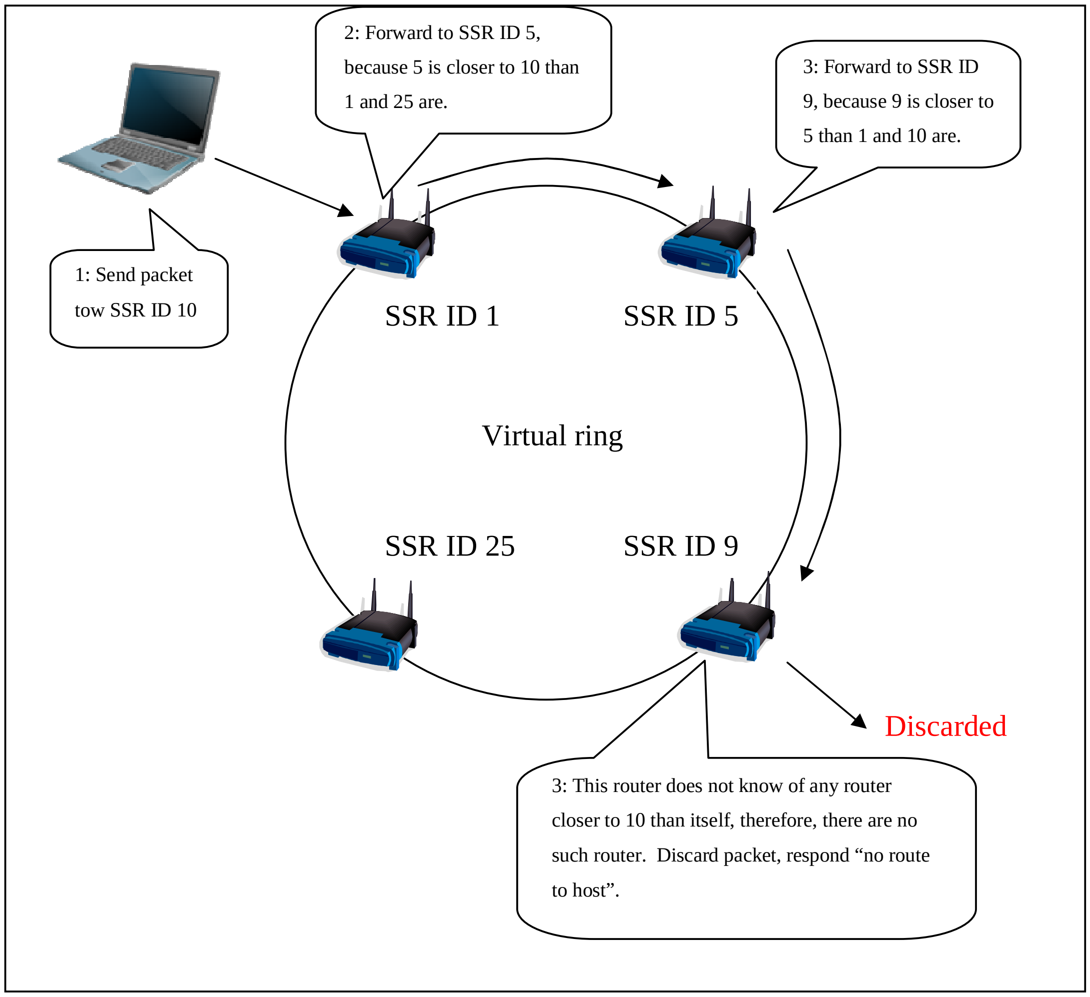

2: Forward to SSR ID 5, because 5 is closer to 10 than 1 and 25 are.
3: Forward to SSR ID 9, because 9 is closer to 5 than 1 and 10 are.
1: Send packet tow SSR ID 10
SSR ID 1
SSR ID 5
Virtual ring
SSR ID 25
SSR ID 9
Discarded
3: This router does not know of any router closer to 10 than itself, therefore, there are no such router. Discard packet, respond "no route to host".

Figure 18: How "default routing" in SSR handles packets to none-existing nodes

When a router does not know any path to the final destination of an SSR packet, it will always forward it to the router with an SSR ID closest to the destination SSR ID. By design, all SSR routers do always know the path to the next virtually preceding and succeeding router [21]. If the router would not know any router that is closer to the destination SSR ID than itself, it is certain that the current router is in fact the closest possible and the destination router does not exist. The packet will be dropped, and – in the Linyphi implementation – the ICMPv6 "no route to host" message will be sent back to the source of the packet. *From this point, this behaviour will be referred to as* ***"default routing"***. The behaviour is illustrated in Figure 18. For illustrative purposes, this figure assumes the SSR nodes *only* know the physical path to their predecessor and successor (the closest nodes on either side in the virtual ring), thus ignoring the nodes may have cached the path to other nodes.

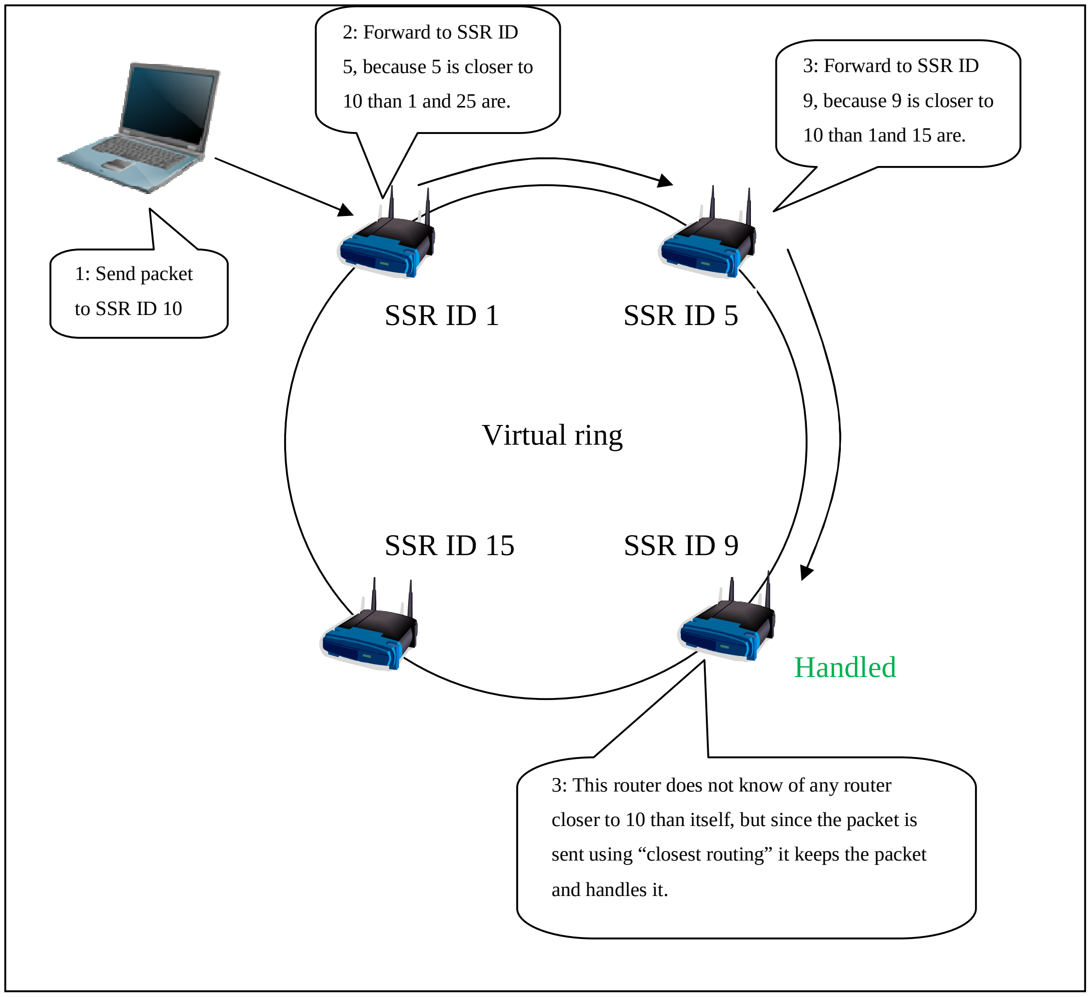

2: Forward to SSR ID 5, because 5 is closer to 10 than 1 and 25 are.
3: Forward to SSR ID 9, because 9 is closer to 10 than 1and 15 are.
1: Send packet to SSR ID 10
SSR ID 1
SSR ID 5
Virtual ring
SSR ID 15
SSR ID 9
Handled
3: This router does not know of any router closer to 10 than itself, but since the packet is sent using "closest routing" it keeps the packet and handles it.

Figure 19: How "closest routing" in SSR handles packets to none-existing nodes

For the DHT functionality, we need to change Linyphi in such way that some packets are treated differently, and instead of being dropped in the first scenario, will be handled by custom code or delivered to the application layer (see section 3.5 for a discussion about that). *This behaviour will be referred to as* ***"closest routing"***, and is illustrated in Figure 19.

SSR contains a set of packet types. There is one single packet type which defines regular data packets (the “Connect” packet type), and several others defining a number of routing messages. What packet type a packet is of, is indicated by a flag in the packet header (as described in section 2.1.4.4).

We cannot simply implement “closest routing” on all “Connect” packets. If so, all such packets which are destined to a non-existing or unreachable router, and are not meant to be handled by our DHT, will still always arrive at some router, and perhaps be received by an application. This may result in unpredictable and erroneous results. For these packets we want “default routing”, so that they are not handled and so that an ICMPv6 “no route to host” packet is sent back to the source.

Therefore, we will implement a new SSR packet type. The new type will have packet ID 14 (`0x0E`), and will be identical to the Connect packet type, with the only difference being the action performed when the exact destination could not be found. A parameter will be appended to the functions and methods in Linyphi and SSR that receive and send packets; when sending packets the parameter specifies what type of routing should be made, and when receiving packets the parameter returns which type of matching that have been done. The parameter is an enumeration called “`TARGET_TYPE`”, and has the `int` values “`TARGET_TYPE_DEFAULT`” (0, default) and “`TARGET_TYPE_CLOSEST`” (1). Other values can be added in the future if so necessary, and the parameter will be put inside a structure (`TARGET_MODE`) to allow other related parameters to be added without further modifications of any function headers. *Note: adding this extra parameter throughout the application may affect the overall performance of Linyphi.*

Another approach is to beforehand inspect the first router’s SSR routing table and directly locate the SSR ID closest to the hash value. However, this is not possible, because there is no guarantee one SSR node is aware of all other SSR nodes. The router with the SSR ID which best match the destination may therefore not exist in the first routers routing table. It is essential that independently of the current location in the SSR network, SIP URI/IP address-pair for one specific SIP URI (the SIP URI is the key in the DHT) are always both stored and retrieved on one, unambiguous location; otherwise they will not form a consistent DHT.

Note however, that when a router has just joined the network, its routing information may be inaccurate until the final paths have been established. Therefore, when packets pass through a newly joined router, the packet might not be forwarded. However, all standard SIP clients send `REGISTER` at regular intervals [17], so any effects of this should only be temporary.

3.4.2 Variation of SSR IDs

Linyphi assigns SSR IDs to be identical to the MAC-48 address of the first interface of the router. Because of the way MAC-48 addresses are assigned to hardware [22], this will ensure that any SSR ID will always be globally unique. However, the SSR IDs will not be well distributed over the address range, as MAC-48 addresses are defined hierarchically. For example, each hardware manufacturer owns a certain range of the value domain and thus hardware from a single manufacturer will have MAC-48 addresses relatively close to each other.

This is not a problem for normal use of SSR, but when using it as a DHT and using the nodes addresses to determine where to store each DHT entry in, the addresses must be well distributed over the address value range to evenly distribute the load put on each node. An example best describes why.

Three of the Linksys WRT54GL routers we have available have the following MAC-48 addresses:

Router 1:	`00:1C:10:52:1D:BF`
Router 2:	`00:1C:10:52:44:56`
Router 3:	`00:1C:10:52:47:B3`

Table 4: MAC-48 of three Linksys WRT54GL routers

When running Linyphi on these routers, the SSR IDs will be identical to these MAC-48 addresses. Note that `00:1C:10` is one of the address ranges owned by “Cisco-Linksys, LLT” [23].

Now, consider some random 48-bit hash values. Hash values are per definition spread over the available range [24].

```
3C:29:F0:7C:15:9D
5A:36:26:61:BC:C9
66:84:10:E2:E9:17
9D:9F:7B:21:4F:EC
9D:A9:22:95:86:65
B2:F3:FB:21:2E:E0
C6:0A:F2:6F:D1:13
CF:78:98:F6:03:04
E1:4F:00:22:25:F8
EA:CF:8D:F7:B8:02
```

Table 5: Ten random 48-bit hash values (ordered)

Even though this sample is too small to be well distributed, these could be hash values of different SIP URIs. As described in sections 3.3.3 and 3.4.1, this would then be the destination of the SSR packets containing SIP user information: Because of the way SSR is designed, and because its mechanism is used by the DHT, each node is responsible for all packets targeted to addresses targeting from the nodes own SSR ID up until all addresses lower than the successors SSR ID. The table below lists on which router the user information associated with these hash values and MAC-48 addresses will be stored.

```
3C:29:F0:7C:15:9D -> router 3
5A:36:26:61:BC:C9 -> router 3
66:84:10:E2:E9:17 -> router 3
9D:9F:7B:21:4F:EC -> router 3
9D:A9:22:95:86:65 -> router 3
B2:F3:FB:21:2E:E0 -> router 3
C6:0A:F2:6F:D1:13 -> router 3
CF:78:98:F6:03:04 -> router 3
E1:4F:00:22:25:F8 -> router 3
EA:CF:8D:F7:B8:02 -> router 3
```

Table 6: Distribution of hash value to SSR ID approximation

We can clearly see that the SIP user entries are not well distributed among the routers, as router 3 has to store all user information. Only the very few hash values between `00:1C:10:52:1D:BF` up until `00:1C:10:52:47:B2` (router 3 was ...`B3`) will be stored on router number 1 and 2. This is less than 256^2 of the available 256^6 addresses.

The problem is also illustrated in the figure below. The three patterns visualize the address range in the virtual ring that each of the three routers cover. The figure is not to scale.

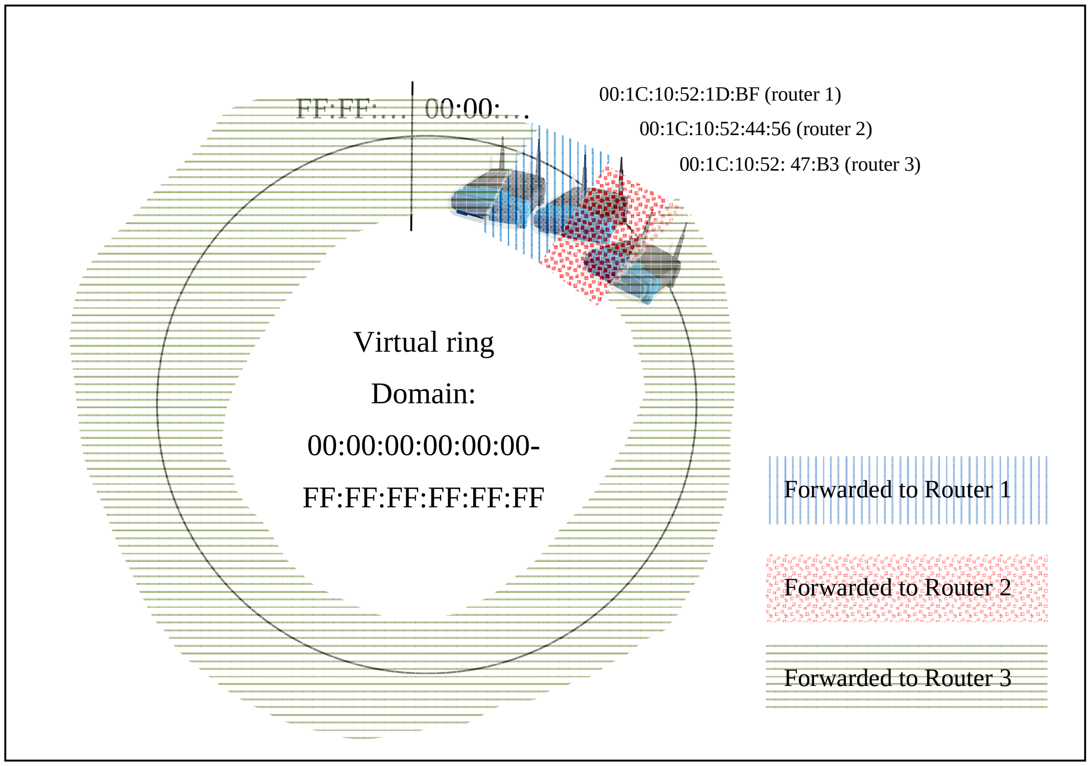


Figure 20: Illustration of the problem with unevenly distributed SSR IDs

Clearly, the MAC-48 addresses of these routers are too close to each other. The problem can be worked around by manually overriding or replacing the factory assigned MAC-48 address of the routers in the network, to some well distributed values.

However, a more lasting solution would be to change the way Linyphi assigns SSR IDs, or how SSR compares SSR addresses to determine what is closer. This should be done in some way which does not affect packets sent using "normal routing".

What we will do is to simply reverse the bytes of the MAC-48 address when it is read from the system, because the end of the MAC-48 is much more random than the beginning. Applying this to our example would give the much more improved distribution as shown below. Incidentally, two of the routers happened to have fairly close final digits, so the distribution is still not very even, but for a larger amount of routers, the distribution should be even better.

Router 1:	`BF:1D:52:10:1C:00`
Router 2:	`56:44:52:10:1C:00`
Router 3:	`B3:47:52:10:1C:00`

Table 7: Byte wise reversed MAC-48 of three Linksys WRT54GL routers

```
3C:29:F0:7C:15:9D -> router 1
5A:36:26:61:BC:C9 -> router 2
66:84:10:E2:E9:17 -> router 2
9D:9F:7B:21:4F:EC -> router 2
9D:A9:22:95:86:65 -> router 2
B2:F3:FB:21:2E:E0 -> router 2
C6:0A:F2:6F:D1:13 -> router 1
CF:78:98:F6:03:04 -> router 1
EA:CF:8D:F7:B8:02 -> router 1
E1:4F:00:22:25:F8 -> router 1
```

Table 8: Distribution of hash value to reversed SSR ID approximation

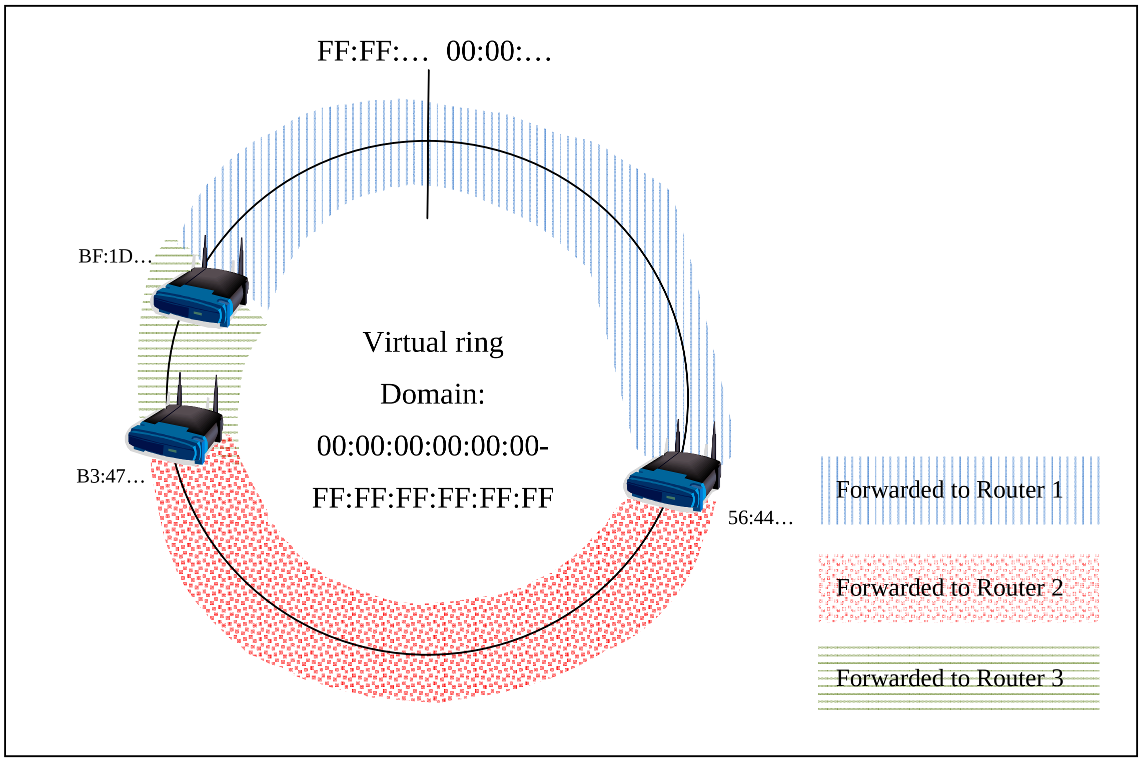


Figure 21: Illustration of the problem with unevenly distributed SSR IDs being solved

3.5 Implementation of new functionality

3.5.1 Motivation

Since we want to run SIP clients with minimal modifications, our SIP and DHT functionality needs to somehow be managed by an implementation running on the routers. This implementation needs to receive and handle all SIP packets destined for the specific router (“default routing”), and to all SIP packets which has destinations most approximate to the specific router (“closest routing”).

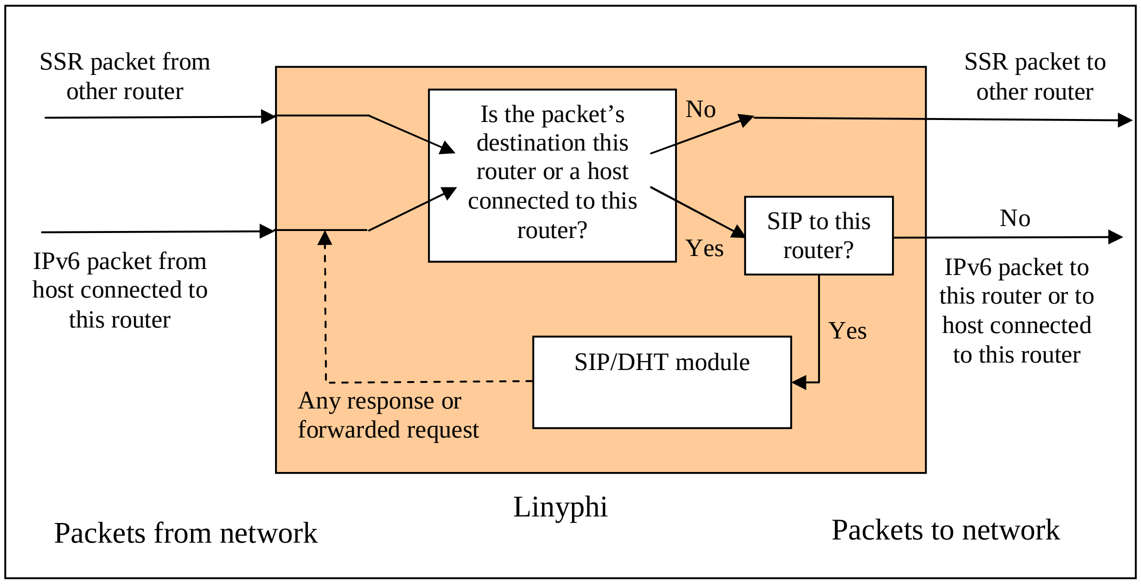
If a packet is delivered to the application layer by Linyphi, we know that the packet should be handled by the current router, independent of routing method; therefore we could easily implement the SIP functionality as a second process running on the router, listening for and *receiving* packets targeted to the SIP socket of the router.

However, this only works one way. For the DHT functionality to work, the implementation also needs to be able to *send* selected packets using our custom type of routing (“closest routing”). To make the implementation work as a separate process, this option must be added to one of the standard transport or network layers, or, we need to implement some inter-process communication and thereby adding unnecessary overhead.

Another option is to use LibIgor [20]. LibIgor is an API which uses Linyphi to act as a transparent P2P layer for any application. LibIgor can hash addresses and send them and use routing similar to our “specific” and “closest” routing methods. This approach was used in [19]. However, the limited documentation makes it hard to see what can actually be done with LibIgor, and there might be a possibility that using this might either create a large overhead or supply insufficient functionality.

Instead, we will implement SIP functionality ourselves as a module inside the Linyphi binary, by adding a class to the Linyphi source code. This allows direct communication “between” Linyphi code and the SIP code, with very little overhead.

3.5.2 Details



SSR packet from other router
IPv6 packet from host connected to this router
Is the packet's destination this router or a host connected to this router?
No
Yes
SIP to this router?
No
Yes
SSR packet to other router
IPv6 packet to this router or to host connected to this router
SIP/DHT module
Any response or forwarded request
Linyphi
Packets from network
Packets to network

Figure 22: Linyphi extended with the SIP module

For all packets destined for the router (either using “default” or “closest” routing), a small function of the module (technically, a method of the SIP object) will inspect the packet and see whether it is a SIP request/response or not and thus if the SIP module should handle it. This is done simply by checking if the destination port of the packet is to the default SIP port, 5060. If so, the packet is handled by the SIP module. If not, the packet is delivered to the application layer as usual. *Note: adding these inspections may affect the overall performance of Linyphi.*

An overview of how packets will be sent to and from the SIP module can be seen in Figure 15 (compare with Figure 7, which shows Linyphi before adding the SIP module). All IPv6 packets leaving Linyphi are inspected, and if a packet is a SIP packet destined for the router itself, it is sent to the SIP module instead of being delivered to the application layer, and is then parsed and handled by the SIP module.

When the SIP module sends a response packet or forwards a request packet, it is simply directed to the same function of Linyphi that any packet arriving from the network is. The reason for this is that most packets departing from the SIP module needs to be routed exactly the same way as other packets arriving at the router - the SIP module cannot easily know if the destination host is connected to the current router or not.

The module will be designed in such way that other modules in the future can be added by adding classes similar to the SIP class. Functions that may be used for other purposes, such as the hash function, will be made as global functions outside the SIP class.

The module will be written in C++ using the "uclibc++" libraries (which also Linyphi use), hence will not be affecting the requirements for running the application.

3.5.3 Hash function

The hash function that will be used is retrieved from [43] and is the following C code:

```
unsigned long hash(unsigned char *str)
{
        unsigned long hash = 5381;
        int c;
        while (c = *str++) hash = ((hash << 5) + hash) + c;
        return hash;
}
```

Noteworthy, it returns an `unsigned long`. We modify it slightly to return a `uint_32` integer, to make it compatible with the "uclibc++" libraries. Also, to be able to compare the hash value (32 bytes) with the MAC-48 addresses (48 bits), we will pad the returned hash value with zeroes in the least significant (the right most) end.

3.6 Scenarios

In this section, we will describe in detail how different scenarios will be implemented.

3.6.1 A SIP user registers its presence in the network

When a user sends a `REGISTER` request, the request is as explained in section 3.2, sent to his/her local router via IPv6 from the SIP client software. Our custom software on the router reads the SIP URI from the request, uses a hash function to calculate a numerical hash value, and sends the request as an SSR packet marked for "closest routing" towards the router which has an SSR ID closest to the hash value, using the routing method explained in section 3.4.1. This router then stores the URI together with the source IP of the request (which therefore must remain unchanged when forwarding), and sends a confirmation back to the source host.

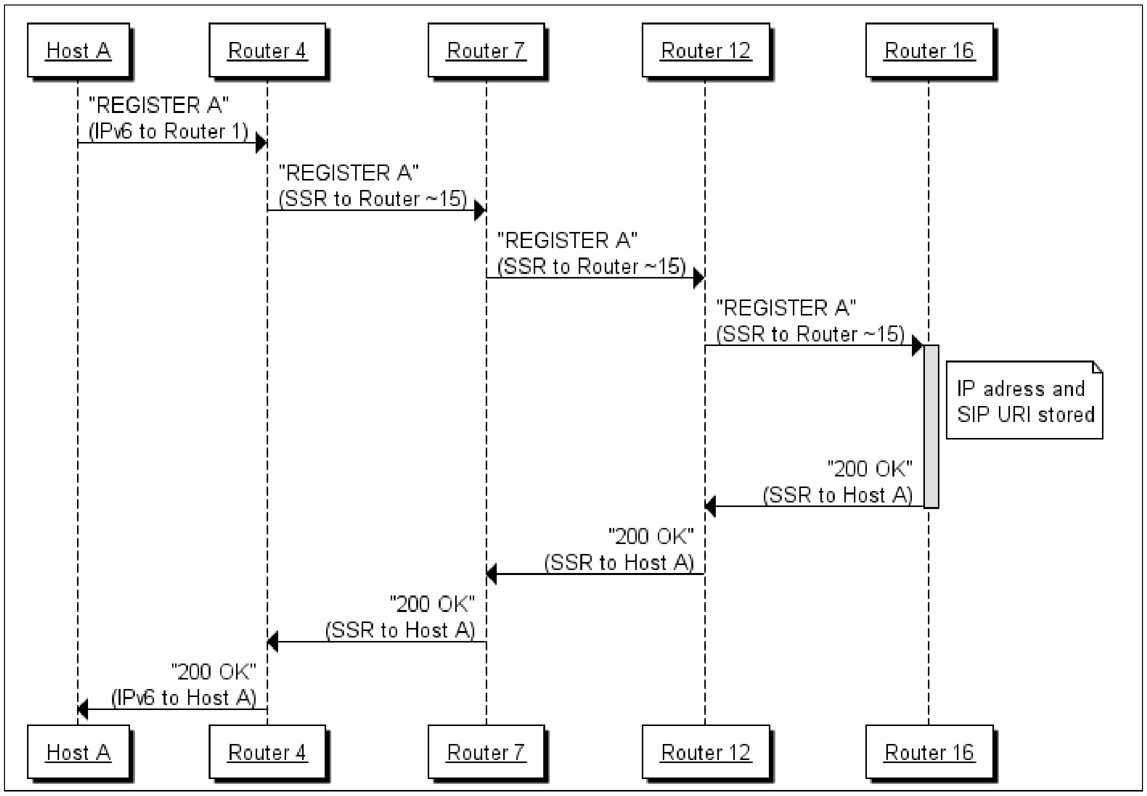

Host A
Router 4
Router 7
Router 12
Router 16
"REGISTER A" (IPv6 to Router 1)
"REGISTER A" (SSR to Router ~15)
"REGISTER A" (SSR to Router ~15)
"REGISTER A" (SSR to Router ~15)
IP adress and SIP URI stored
"200 OK" (SSR to Host A)
"200 OK" (SSR to Host A)
"200 OK" (SSR to Host A)
"200 OK" (IPv6 to Host A)

Figure 23: Sequence diagram of SIP registration using our solution

The sequence diagram in Figure 23 shows a network containing routers with SSR IDs 4, 7, 12 and 16. The hosts and the routers are physically connected as in the depicted order, that is, Host A can only communicate with Router 4, Router 4 can only communicate with Host A and Router 7, Router 7 can only communicate with Router 4 and 12, and so on. This topology is used here to show how packets are sent over a given path; if the routers were connected more randomly there would be several available routes and the shortest routes would be approximated by the SSR protocol. The user at Host A has a SIP URI which results in a hash value of 15. The user at Host A registers on the network.

3.6.2 A SIP user invites another SIP user in the network

When a user invites another user, the `INVITE` request is (just like a `REGISTRATION` request) forwarded by the first router to the router responsible for storing the location of the invited SIP user, based on the hash value of the SIP URI of the invited user. The invitation is then forwarded further to the host to invite.

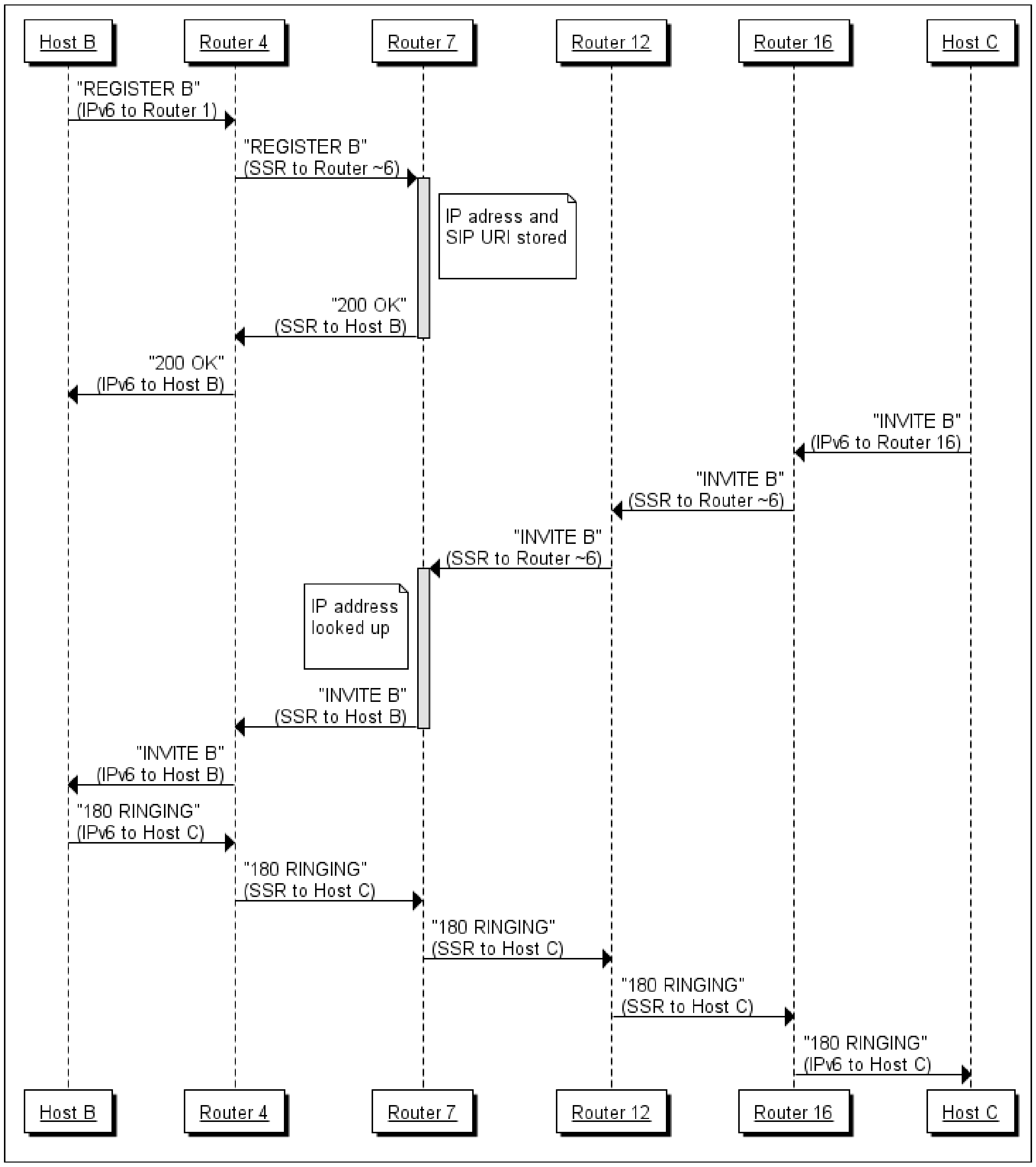

Host B
Router 4
Router 7
Router 12
Router 16
Host C
"REGISTER B" (IPv6 to Router 1)
"REGISTER B" (SSR to Router ~6)
IP adress and SIP URI stored
"200 OK" (SSR to Host B)
"200 OK" (IPv6 to Host B)
"INVITE B" (IPv6 to Router 16)
"INVITE B" (SSR to Router ~6)
"INVITE B" (SSR to Router ~6)
IP address looked up
"INVITE B" (SSR to Host B)
"INVITE B" (IPv6 to Host B)
"180 RINGING" (IPv6 to Host C)
"180 RINGING" (SSR to Host C)
"180 RINGING" (SSR to Host C)
"180 RINGING" (SSR to Host C)
"180 RINGING" (IPv6 to Host C)

Figure 24: Sequence diagram of SIP registration and invitation using our solution

The sequence diagram in Figure 24 shows a network containing routers with SSR IDs 4, 7, 12 and 16. The hosts and the routers are physically connected as in the previous section. The user at Host B has a SIP URI which results in a hash value of 6. The user at Host B registers on the network. A user at Host C then tries to make a phone call to the user at Host B, knowing only the Host B user's SIP URI. Results (“`200 OK`”, assuming the user accepts the call), media transfer and call teardown has been left out of this diagram.

3.6.3 A new router joins the network

As the nodes participating in a mesh network may change, consideration must be made to how to react to these changes. Note that our implementation assumes that the network is static. However, it will be possible to add support for handling the addition or removal of a new router to our implementation. Below is described what needs to be implemented.

A new router that joins the network might have an SSR ID closer to a certain hash value than another. For example: The network has routers with SSR IDs 1, 5 and 10, and the hash value of `alice@kau.se` is 7. Therefore, the location information of Alice is stored on router 5. Then, a new router with the SSR ID 6 joins the network. All invitations meant for Alice is now redirected to router 6, while her location information is still stored on router 5. To fix this, the following functionality must be implemented and performed on a router, every time the router receives information of a previously unknown router:

```
For each entry in the DHT table containing SIP URI/IP entries:
    If the hash value of the SIP URI is closer to the joined router's SSR ID than its own:
        Send the entry to the new router as a SIP REGISTER message to the new router
            (The new router will either store the entry and reply with a confirmation message,
             or forward the message back to the current router)
For each received successful confirmation:
    Remove location information from current router
```

3.6.4 A router leaves the network

If a router that contains location information is disconnected or disabled, the location information stored on that router would be lost.

If the disconnection is done under controlled circumstances, so that Linyphi can be made to execute some specific “shutdown” code before shutting down, the locations of all SIP users may be simply forwarded to its successor or predecessor (depending on which one is closest to the hash value of each SIP entry in the SIP URI/IP address table).

However, if the router is disconnected due to a network failure or any other unexpected error, the location information is lost or unreachable. This problem is only temporary however, as SIP clients send periodic registration messages [17].

It would be possible to eliminate this problem to a great extent by always making the router with the SSR ID closest to the hash value of the SIP URI share its location information with its predecessor and successor, depending on which one is closest to the hash value of each individual SIP entry. Note that the predecessor/successor should *not* share that location information further, as it would effectively spread all location information over the entire network.

This implementation is not trivial and would add overhead, but it would make sure there would always be at least one reachable location information entry even if one single router is disconnected.

Another problem occurs if the removed router is the physical link between two networks. If this happens, it is unavoidable that hosts in the two separate networks will be unable to call each other.

Just after such an event some hosts will be unreachable even to hosts within the same network, if their location information was stored on a router in the other network. However, since standard SIP clients send periodic registrations, this problem will only have temporary effects.

3.7 Scope

The items described in this section will need to be implemented to cover all scenarios for making our extension usable in practice; however, due to the size of such a project, we must focus on implementing basic functionality for testing and comparison with existing methods. The sections below describe what we initially will and will not implement.

3.7.1 Performance

The primary objective is to make the new functionality reliable enough to prove the concept, efforts to improve performance in second-hand but will be done if time permits.

3.7.2 UDP and TCP

Compared to TCP, the UDP protocol has higher throughput at the cost of detecting lost packets [18]. This makes UDP most suitable and most common for real time media transportation, such as VoIP, and SIP is designed to function over UDP.

We will focus on implementing all functionality for SIP packets sent using UDP. The result of Linyphi receiving a TCP SIP packet destined for the router will be undefined;

however we will attempt to prepare the new code for such an implementation and it will be implemented if time permits.

3.7.3 Internet connectivity

As mentioned in section 3.2, the SIP module should inspect whether the SIP request is destined for the local network or the Internet by looking at the destination SIP URI domain. This will only be implemented if time permits; instead we will assume all packets are destined for a host within the local network, regardless of their specified domain.

3.7.4 Mobility

We will assume that no changes will be made to the topology, that is, the location of each router does not change, and no router joins or leaves the network after it has been set up. In practice, this means that the functionality described in section 3.6.3 and 3.6.4 will not be implemented (unless time would permit).

3.7.5 SIP specification compliance

We will focus on making the SIP functionality work with the SIP client we will use for testing, SofSip-Cli. All SIP requests except "`REGISTER`" will be forwarded to the destination specified in the SIP header. Note however, that this should be sufficient for other implementations than SofSip-Cli, but some responses and requests from our implementation may break the rules defined in the SIP definition [17].

The SIP URI table (the DHT table) will be very simple and entries will be stored in the router's RAM until Linyphi is terminated.

3.7.6 Authorization and security

No authentication or encryption functionality will be implemented. Also, there will be no way to ensure that any SIP URI is unique.

3.7.7 Error handling

Unless time permits, we will not implement any handling of or recovery from unexpected errors (such as malformed packets or insufficient memory).

4 Result and evaluation

4.1 Introduction

In section 4.2 and 4.3, we will describe the problems encountered during the realization of our solution. In 4.4 through 4.6, we will then evaluate our work, by looking at how much of the functionality we managed to implement and other decisions and plans made in section 3. In 4.7 through 4.9 we will evaluate the performance of our solution.

4.2 Versions of Linyphi

When we started working on the project, there was only one version of Linyphi publically available, 0.1. However, as described below in section 4.5-0, we were unable to achieve performance comparable with that described in [2]. After contact with the authors of Linyphi, we learned that there are four relevant versions of Linyphi:

- An initial version: The version used in [2].
- 0.1: The initial version above was modified to be compatible with the ARM platform, but as a side effect of these changes, performance on the Linksys MIPSEL devices was affected negatively. This was unexpected, and the initial version is no longer available. This version does not compile for OpenWRT Kamikaze. This version was the only available version of Linyphi during most part of our work, so it is this version we have added the extension to.
- 0.2: Has similar performance as 0.1, but does compile on OpenWRT Kamikaze, and has some other modifications (which do not affect performance significantly).
- A newer version, currently under development: The authors of Linyphi are currently in the process of making a new version of it, completely from scratch. This new version is expected to have greatly improved performance.

Due to the substantial work it would require, we have not attempted to adapt our extensions for 0.2. It should be kept in mind that some issues discussed in the following sections may already have been resolved, or might be resolved in the upcoming version.

4.3 Encountered problems

In this section, we discuss problems we did not expect to encounter, and how we tried to solve or circumvent them.

4.3.1 Lack of IPv6 capable SIP software

The requirement of IPv6 compatible SIP software is crucial to the usability of our work. However, the lack of such proved to be a problem. These were the most promising and usable IPv6 compatible SIP applications we were able to find:

- **SofSip-cli [14]:** Fully functional SIP client that uses the Sofia-Sip library [13] (developed by Nokia) and the GStreamer library [15]. This is the client we used for all our testing. However, as it is console based, it is practically unusable for most end users. Also, the actual media transfer is done over IPv4.
- **Linphone [12]:** User friendly graphical client for Linux and Windows with IPv6 support, however, some technical problem prevented us from using it (it crashed upon start-up). Linphone (Linux + phone) should not be confused with the previously mentioned Linyphone (Linyphi + phone).
- **KPhone [16]:** Graphical SIP client, of which different versions [10, 11] have been modified to support IPv6. We have not been able to compile or successfully run any of these under Ubuntu 7.10.

4.3.2 Linyphi and the Linux kernel

Linyphi is a stand-alone application, and does not use any kernel modules. It receives packets from the network only using system calls and reading from raw network sockets. However, the kernel does some things independently and regardless of the actions performed in Linyphi, such as responding to inappropriate packets. Since the kernel finds some ICMPv6 routing packets inappropriate (which are actually used by Linyphi), the kernel will respond with other ICMPv6 packets. Linyphi prevents these kernel responses from reaching the network by automatically setting up iptables to drop certain outgoing ICMP packets.

When creating our extension, we ran into a similar problem. When sending an IPv6 packet directly to the router (that is, not to one of the hosts connected to the router), the packet was delivered by the kernel “up” to the application that is listening on the destination port of the packet. If no such port was found, the kernel responded with an ICMPv6 message. This is normally correct and appropriate, however this was also done when Linyphi itself discarded or handled the packet, in our case by our SIP extension: That is, even though a SIP packet was forwarded or handled properly by Linyphi and our extension, it was also delivered to the local application layer. As no application was registered to listen to the packets destination port, an ICMPv6 packet was returned to the SIP client, making it believe the delivery failed while the packet was actually handled properly by Linyphi.

We considered using iptables to block the resulting ICMPv6 messages, but for our problem, we found it more suitable to simply create a “dummy listener”: While Linyphi is running, it is itself listening to the default SIP port (5060), and quickly discards all packets delivered to it from the kernel (remember that it uses other, lower-level, methods to retrieve the packets from the kernel).

This approach also prevents other, possibly conflicting, SIP proxies or registrars to listen to the same port, and it is also not sensitive to accidental modifications of iptables while Linyphi is running.

4.3.3 No media transfer

Unfortunately, due to the lack of IPv6 compatible SIP clients, we have been unable to successfully establish media transfer between hosts over Linyphi. This is because the only SIP client we have successfully configured and used, SofSip-cli, only uses IPv6 during session initiation setup (when SIP is used). After the session has been successfully established using SIP and IPv6, the client leaves it up to the media module to establish a media connection. We were unable to find a compatible media module for IPv6. We have confirmed that the connection attempts for media over IPv4 are done between the correct hosts using their correct IPv4 address, and we assume that if IPv4 would be enabled, media would transfer properly. Once the call is being ended by the caller or callee, it is torn down properly by the SIP protocol.

Note that this is a client related issue; everything related to SIP and Linyphi in this matter appears to be functioning properly.

4.3.4 Linyphi robustness and compatibility

As described in section 4.2, we added our extension to Linyphi 0.1. During the testing phase, a newer version (0.2) of Linyphi was made available by its authors. We have worked very little with 0.2 compared to 0.1. The following problems were encountered in 0.1, and may very well have been resolved in 0.2.

There are two relevant releases of OpenWRT: White Russian and Kamikaze. White Russian is based on Linux kernel 2.4 while the newer Kamikaze is available with both Linux kernel 2.4 and 2.6. [5]

While it would be most appropriate to use the latest available version of OpenWRT, we were unable to build Linyphi 0.1 for the Kamikaze version of OpenWRT. Linyphi 0.2 *does* compile for Kamikaze.

Linyphi did on a few occasions (in some cases after running uninterrupted for several days, sometimes sooner) crash with a memory allocation error. We observed this behaviour also in the unmodified version (that is, before any modifications were made to it in this project).

We also noticed that if the router is overwhelmed with packets in such way that a large amount of packets arrive faster than they are forwarded (very noticeable when packets arrive from a host connected by 100 Mbps wire, with a destination being a host only reachable through a 54 Mbps wireless connection), the kernel's transmission buffer would quickly be filled. While this is normal, Linyphi does not handle this situation properly but crash when a sending error occurs due to the overfull buffer. We observed this very quickly using SIPp [8], when establishing about 3-4 connections or more per second (more connections made it crash quicker).

4.3.5 No change in SSR ID assignments

In section 3.4.2, we described that because of the way Linyphi assigns SSR IDs for the different nodes, IDs often do not spread appropriately to form a well balanced DHT. We also described how to solve this problem.

We attempted to do it according to these plans. However, the required changes were more complex than first expected, and due to time constraints; we decided to use the work-around also described in 3.4.2, which is to manually override the MAC-48-addresses on the routers.

4.4 Evaluation of functionality

In section 3.7 we discussed items that we in beforehand decided to implement only if time would permit. Only limited time was spent to make our extension less sensitive of malformed or unusually formed SIP packets, and to optimize the performance of some of the most complex functions.

In effect, the extension needs to be developed further and all items in section 3.7 should be implemented before our work can be practically useful.

Preferably, also the change in SSR ID assignments that we failed to implement (see section 4.3.5) should be implemented on the routers.

4.5 Performance evaluation method

As described in section 4.2, before gathering any meaningful data, we saw clear indications that Linyphi did not perform nearly as well as described in [2]. We originally planned to compare Linyphi with our SIP extension against other solutions, primarily AODV. However, the current performance of Linyphi made the first planned tests meaningless, so we abandoned them and planned a new series of tests. In the sections below, we describe both the abandoned test plan and the new, revised test plan.

4.5.1 Original (abandoned) test plan

Initially, we intended to measure the performance of our work by comparing call setup delays on the Linksys WRT56GL routers, using our modified version of Linyphi with a standard SIP server running on a host in an IPv4 AODV. The tests would be made using four hosts connected to four routers, as described in the figures below. Different tests would then be performed to evaluate the performance and scalability of Linyphi.

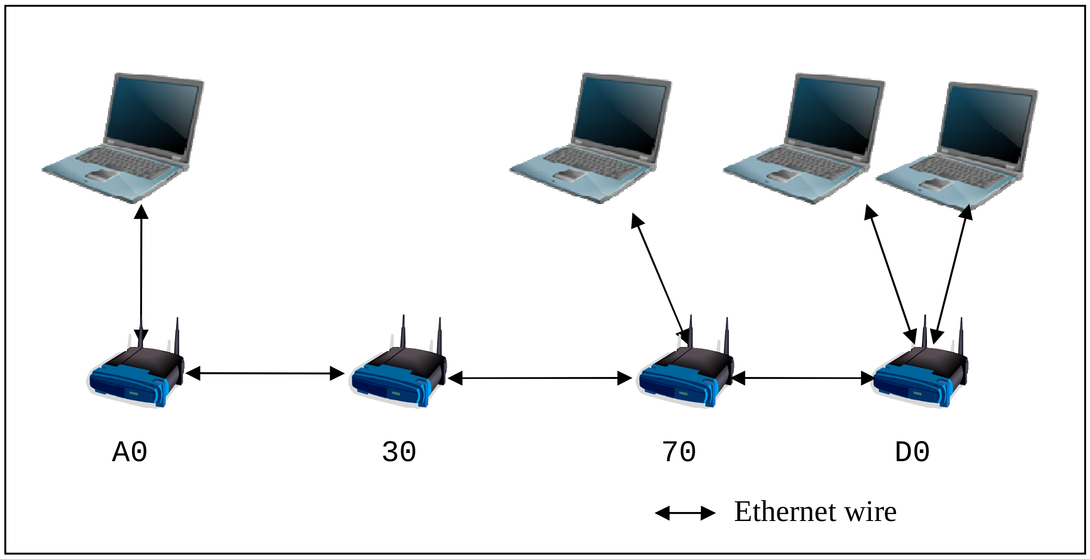


Figure 25: Physical test setup (abandoned). Numbers under routers indicate SSR IDs.

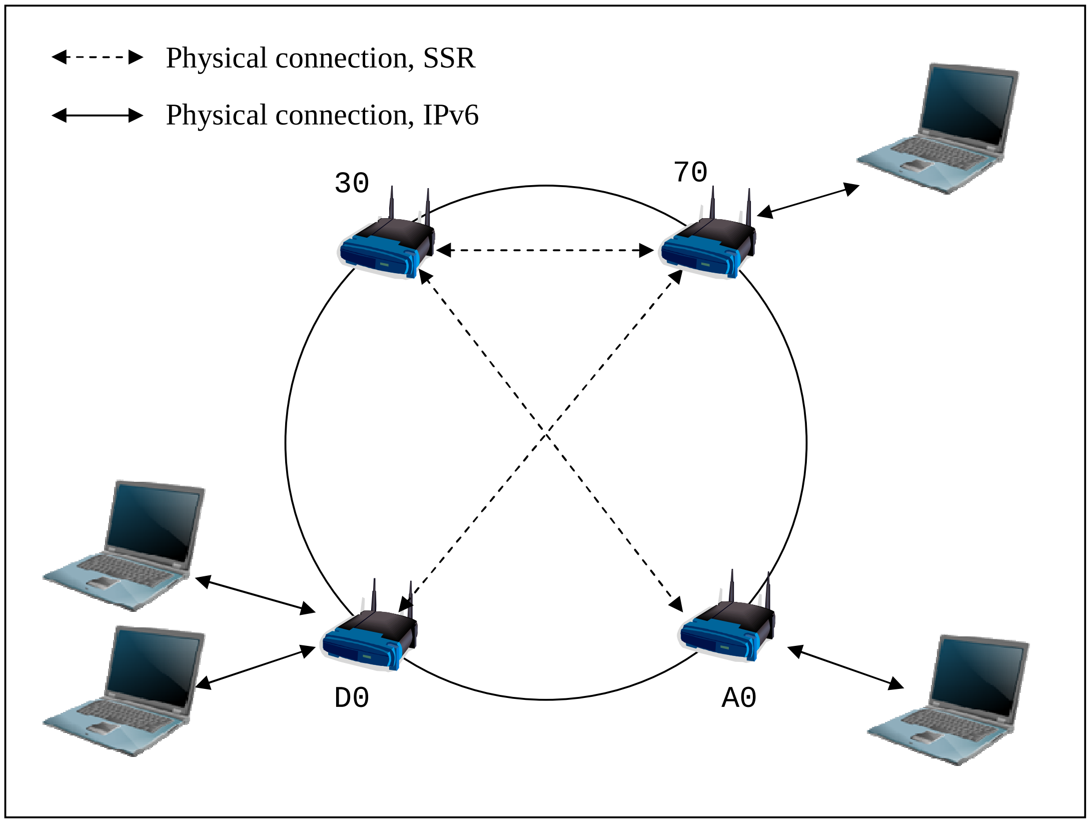


Figure 26: The virtual SSR ring in the abandoned test setup

We intended to test, for both SSR and AODV:

- Round trip time of echo request/response from host at `A0` to `D0`
- SIP `REGISTER` from host at `A0` to router `30`, `70`, `A0` and `D0` (to measure scalability)
- SIP `INVITE` from host at `D0` invite host connected to `A0` and registered at `D0`
- SIP `INVITE` from host at `D0`, to host at `A0`, registered at `A0`
- SIP `INVITE` from one host at `D0`, to other host on `D0`, registered at `A0`
- SIP `INVITE` from one host at `D0`, to other host at `D0`, registered at `D0`

For AODV, we would set up a host running a SIP proxy/registrar to replace the routers registration services. All tests would be performed both with and without background traffic.

Note that for our SIP extension, it is not normal to choose which router to register on: The first router, which is set as SIP proxy in the SIP client software, will forward the registration, and the routers will continue to forward it until it arrives at the router which SSR ID is closest possible to the hash value of the SIP URI. By deliberately using a set of SIP URIs of which we have calculated and know the hash values for, we can ensure that user locations will be stored on routers of our choice. For example, to test a registration on router `30`, we register a SIP URI which we know have a hash value virtually closest possible to `30`.

The figures and lists above are simplified. For example, router `A0` actually has one SSR ID and two MAC addresses: `A0:A0:A0:A0:A0:27` is the SSR ID and MAC address of the wired interface, while `A0:A0:A0:A0:A0:29` is the MAC address of the wireless interface. The explanation for these particular MAC-addresses are as follows:

- The last two bits in a MAC-48 address' first byte determine if the address is unicast (`0`) or multicast (`1`), and globally unique (`0`) or locally administered (`1`), respectively. If the unicast/multicast bit is set to `1`, the MAC-48 address may cause problems. To make the generation of MAC addresses as easy as possible, we always set the last four bits to `0`. The result is that we set the first byte to `X0`, where `X` is any value.
- For easy generation of MAC addresses, we set the following four bytes to the same as the first byte.
- We set the last byte to `27` and `29`. These values are selected randomly. (In fact, the default addresses on one of our Linksys routers ends with `27` and `29`.) It seems to be common practice by Linksys to assign default addresses where the last byte of the MAC-48 are numerically odd (last bit is `1`), and the addresses of the wireless interfaces is two bits higher than the wired interfaces. (Each router has one wireless and one wired interface). To avoid problems we kept this practice. To more easily remember the addresses we set them to `27` and `29` on all routers regardless of the previous bytes.
- The SSR ID assigned by Linyphi is the MAC-48 address of the wired interface (which ends with `27`), simply because it is the first interface in the device. Recall that the only reason Linyphi reads the MAC-48 address, is to get a value which is ensured to be globally unique. [2]

The original MAC addresses and SSR ID assignment are unusable because the first 32 bits are the same for all our four routers (`00:1C:10:52`), which is a problem we described in section 3.4.2. There, we also describe a possible solution, which we describe why we failed to implement in 4.3.5. Therefore, we must manually override the MAC address of the routers, by setting an NVRAM variable which overrides the physically assigned MAC address. This must be done during every start-up of the routers, before the network interfaces are brought up, or the original MAC-48 address will be used and cannot be modified. Therefore this must be done in a start-up script on the router. (The NVRAM is a memory space in the non-volatile flash memory of the router, which is used to store most of the routers configuration in the form of a large amount of key-value pairs. The “`nvram`” shell command is used to set and get these variables. Note that use of the NVRAM has been abandoned in OpenWRT Kamikaze, in favour of less complicated configuration files.)

4.5.2 Revised test plan

With the current poor performance of Linyphi on the MIPSEL routers, a comparison with any other protocol (such as AODV, as intended) would not be meaningful. Instead we will perform tests which will attempt to show that:

1) The current performance is well below what is acceptable for transferring VoIP
2) The poor performance can be solved by using faster hardware or by optimizing the source code

We will also investigate if:

3) Our modifications have penalized the performance of non-SIP traffic over Linyphi.

This will be done through four tests, each described in the sections below.

In all tests, everything will be connected *by wire* to avoid any interference and to maximize the performance. The network will be completely free of any other load.

4.5.2.1 The reference test

This test alone will partly show if Linyphi currently perform sufficiently for VoIP. All other tests will be compared to this test, and all other test setups will be based on this one.

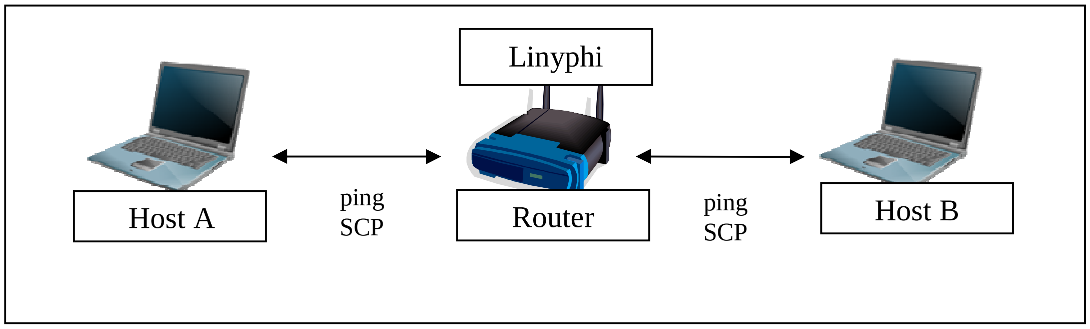


Figure 27: Reference test; unmodified Linyphi running on one Linksys router

In this test, we will connect two hosts to a Linksys router running an unmodified version of Linyphi 0.1. The hosts will be connected to separate virtual interfaces on the router, enabling Linyphi to handle the packets sent between the hosts.

The average of 100 ICMPv6 echo (“ping”) requests will be used to measure the transmission delay, here defined as the time passed between sending a packet at the source host, until receiving it at the target host. In voice communication, this is *roughly* the delay until a spoken sound is heard by the remote person (“mouth-to-ear-delay”). (It is an approximation because in actual voice communication, also microphone, capture and encoding/decoding processes increase the actual delay.) ITU-T claims in [3] that the mouth-to-ear-delay should be lower than 400 milliseconds to be acceptable for voice communication by most users. The round trip times the ping utility will calculate is the total delay from sending an echo request until receiving a response. The transmission delay should be approximately half of the round trip time.

The average transmission speed (or bit rate) will be measured by repeatedly (5 times) sending a large file (approximately 700 MB) between the hosts using SCP. In voice communication, higher transmission speed allows increased sound quality. Deciding a minimum value for voice communication is difficult, as the required bandwidth is dependent on which codec is used to compress the audio. There is a codec which requires only 800 bits/second to transfer (or store) understandable speech [6]; however it is doubtful the sound quality is acceptable by users. On the other hand, today’s standard “analogue” phones use not more than 64 Kilobit/second lines [7], which therefore surely can be regarded as acceptable bandwidth.

4.5.2.2 Testing scalability

With this test, we intend to show if Linyphi in its current state is capable of VoIP.

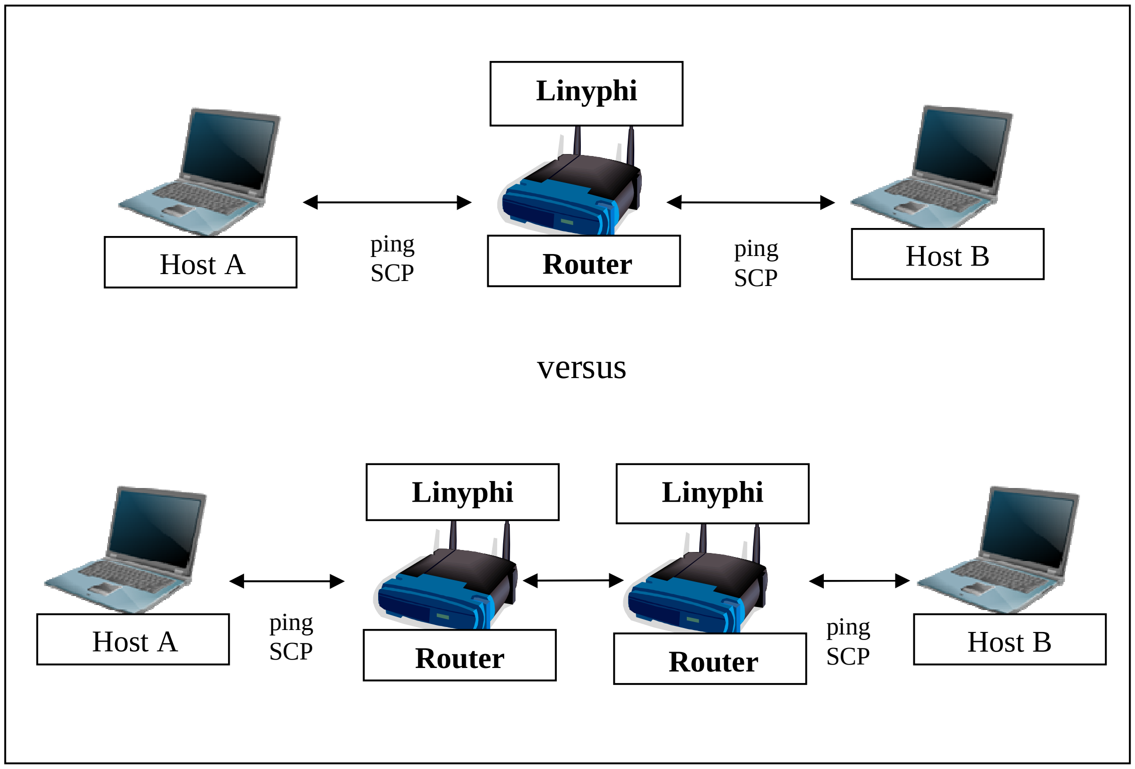


Figure 28: Scalability test; unmodified Linyphi running on two Linksys routers

In this test, we will perform the reference test again, but add one (or more) router(s) in serial. We will compare the results with the reference test to see how performance is affected by additional hops over several routers.

4.5.2.3 Testing processing power

The purpose of this test is to see if adding processing power to the router and/or optimizing the source code can solve the current performance problem.

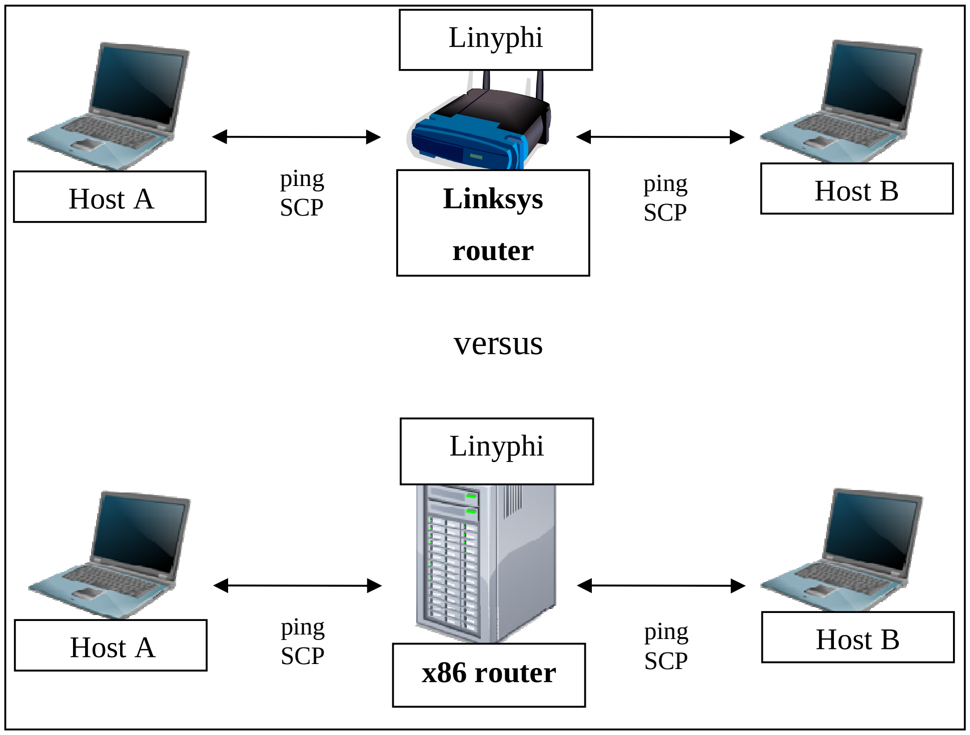


Figure 29: Processing power test; Linksys router versus homemade x86 router

In this test we will repeat the reference test, but will replace the Linksys router. The new router will have similar network capabilities but have more processing power. We will then compare the results with the reference test.

Before we explain the details of the new router, remember that the WRT54GL router, as well as many other routers, is essentially a small computer with a number of wired network interface cards each connected to a switch, and one wireless network interface card [4]. Knowing this, we will build our own router by using a standard x86 PC. It will contain two 100 Mbps network interface cards, thus having the same network performance as the Linksys router, but will have far more powerful processing capabilities. If transmission delay and speed is higher when using our home made router than in the reference test, we know that the low performance of Linyphi on the Linksys router is due to insufficient processing power, and can be solved either by using a more powerful router or by optimizing the Linyphi code. The same unmodified version of Linyphi as in the reference test will be used, but it will need to be recompiled for the different processor architectures (x86 for the home made router and MIPSEL for the router).

4.5.2.4 Testing our modifications

The purpose of this test is to find whether our modifications have affected the performance of Linyphi for non-SIP traffic.

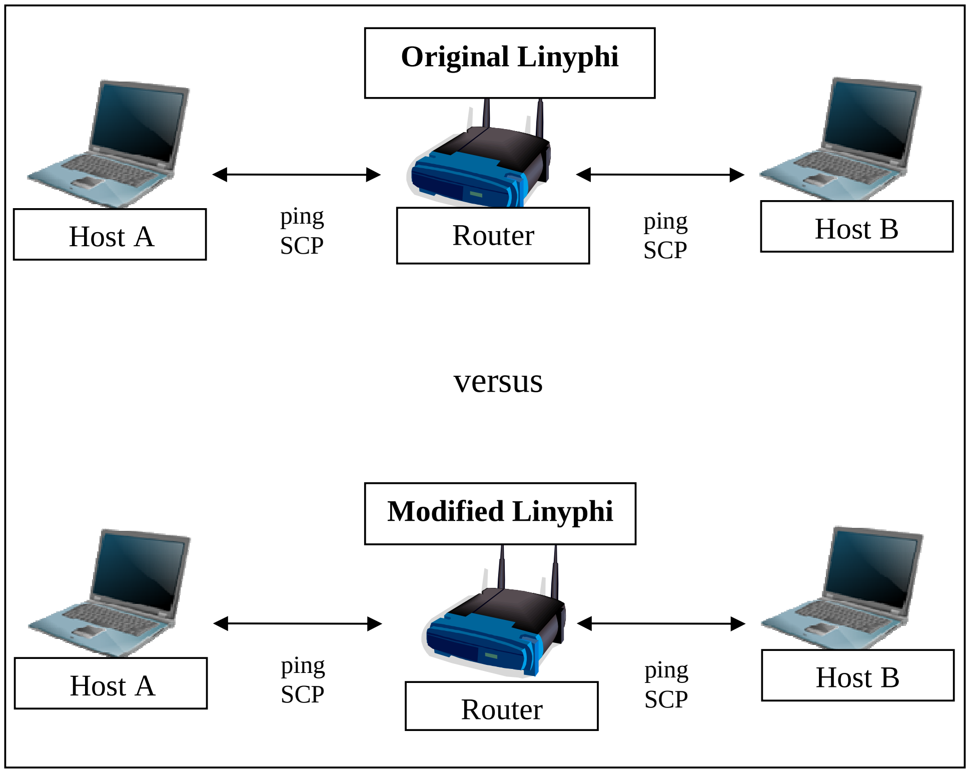


Figure 30: Testing our modifications; Original Linyphi versus modified Linyphi

In this test, we will repeat the reference test, but replace the unmodified Linyphi with a version containing our modifications as described in previous chapters. (Except for the different Linyphi, the test will be identical to the reference test.) We will then compare the results of this tests with the ones received from the reference test.

Note that in all other tests than this, the original, unmodified, Linyphi 0.1 will be used.

4.5.2.5 Equipment and configuration

This section lists the configuration of the hosts and routers used in our tests.

Host A: Intel Pentium Dual-Core @ 2 x 1,600 MHz (x86), 1,024 MB RAM, 1 Gbps Atheros wired NIC (as there are no other 1 Gbps interfaces in the network, this device will always operate in 100 Mbps mode), Ubuntu 7.10 (Gutsy Gibbon), using kernel 2.6.22

Host B: Intel Pentium 4 @ 1,700 MHz (x86), 512 MB RAM, 100 Mbps LiteOn wired NIC, Ubuntu 7.10 (Gutsy Gibbon), using kernel 2.6.22

Linksys WRT54GL routers [4]: Broadcom 5352EKPB @ 200 MHz (MIPSEL), 16 MB RAM, 5 x 100 Mbps wired ports, 54 Mbps 802.11b/g wireless interface (but only the wired interfaces will be used during our tests), OpenWRT 0.9 (White Russian. This is not the latest version - Kamikaze 7.09 is more recent [5] - but the latest version is currently incompatible with the version of Linyphi we have used.) The wired LAN ports are divided into two virtual interfaces, to enable Linyphi to route packets between them (the router hardware routes packets itself within the virtual interfaces).

Homemade x86 router: Intel Pentium III @ 733 MHz (x86), 384 MB RAM, 100 Mbps 3Com wired NIC, 100 Mbps Realtek wired NIC, Ubuntu 7.10 (Gutsy Gibbon), using kernel 2.6.22

4.6 Performance results

The table below lists the performance results. The items in the table are described below it.

See the appendix for complete measurement tables.

	Reference	**Scalability (two routers)**	**Processing power**	**Our modifications**
Minimum round trip time	1.628 ms	9.378 ms	0.414 ms	1.742 ms
Average round trip time	1.695 ms	9.532 ms	0.617 ms	1.810 ms
Maximum round trip time	2.748 ms	10.351 ms	7.031 ms	1.888 ms
Calculated average transmission delay	**0.8475 ms**	**4.766 ms**	**0.3085 ms**	**0.9050 ms**

	Reference	**Scalability (two routers)**	**Processing power**	**Our modifications**
Minimum transmission time	564 s	3500 s	82 s	614 s
Maximum transmission speed	1.24 MByte/s	0.200 MByte/s	8.54 MByte/s	1.14 MByte/s
Maximum transmission time	565 s	3511 s	97 s	615 s
Minimum transmission speed	1.24 MByte/s	0.199 MByte/s	7.22 MByte/s	1.14 MByte/s
Average transmission time	565 s	3507 s	90 s	615 s
Average transmission speed	**1.24 MByte/s**	**0.200 MByte /s**	**7.78 MByte/s**	**1.14 MByte/s**

Table 9: Performance measurements

Minimum round trip time: The shortest round trip time measured of the 100 echo responses, as reported by the ping utility.
Average round trip time: The average round trip times (RTT) of all 100 echo responses as reported by the ping utility.
Maximum round trip time: The longest round trip time measured of all 100 echo responses, as reported by the ping utility.
Calculated average transmission delay: Measured by dividing the *average round trip time* by 2.
Minimum transmission time: The shortest time it took to transfer the 700 MB file out of 5 repeated runs.
Maximum transmission speed: Transmission speed of the minimum transmission time (length of file in bytes divided by the *minimum transmission time* in seconds)
Maximum transmission time: The longest time it took to transfer the 700 MB file out of 5 repeated runs.

Minimum transmission speed: Transmission speed of the maximum transmission time (length of file in bytes divided by the *maximum transmission time* in seconds)
Average transmission time: Average transmission time of the 5 transfers.
Average transmission speed: Average transmission speed of the 5 transfers (length of file in bytes divided by the *average transmission time* in seconds

The length of the transferred file was 734,275,584 bytes.

4.7 Performance conclusions

4.7.1 Scalability

We can clearly see that even though the performance in the reference test may be sufficient for VoIP, and even so in the test with two routers, but the system does not scale. Only a fraction of the available bandwidth is utilized over only one single router.

Bandwidth utilization = Average transmission speed / Theoretical maximum transmission speed

Using 100 Mbps (mega bits per second) as theoretical maximum transmission speed and the average transmission speed of the reference test in the formula above, we observe a bandwidth utilization of 9.92%. (Note that even under ideal conditions, 100% bandwidth utilizations cannot be reached as some of the bandwidth is used for Ethernet headers, error detection, etc.)

When using two routers, only a "fraction of the fraction" of the bandwidth is utilized. If using the average bandwidth in this scenario (0.200 MByte/s) in the formula as above, we observe a bandwidth utilization of 1.60%.

Remember that these tests were performed in ideal conditions: A fast, wired, local network connection with no other network load.

4.7.2 Processing power

When using the more powerful router, Linyphi achieve performance so good that we suspect other factors, such as the read/write speed of the hard disk drives or even limitations of the network may be bottle necks.

The only factor changed here compared with the reference test was the hardware: The bandwidth to the connected devices was still 100 Mbit/s, and the version of Linyphi is identical, except it was for compiled for the x86 instead of the MIPSEL architecture. Effectively, the only thing different between this custom built router and the Linksys routers, is the speed of the processor and the memory.

As Linyphi performs excellent on this more powerful router, we can conclude that the processing power of the Linksys routers is the bottleneck and the cause of the low performance seen in the reference test. Thus, to make Linyphi work on the Linksys routers, its source code needs to be modified or optimized to become more suitable for the Linksys devices.

4.7.3 Our modifications

Unfortunately, there is a slight performance penalty when using our modified version of Linyphi. Average transmission delay was increased from 0.8475 s in the reference test to 0.9050 s (6.8% longer) in the test where our modified code was used. Average transmission speed was decreased from 1.24 MByte/s to 1.14 MByte/s (8.8% slower).

Recalling previous sections, there were two modifications we made to the original source code which may affect the performance of Linyphi:

- In section 3.4.1 we described a method requiring an extra parameter to be passed around, which is used to detect if an SSR packet was sent using "closest" routing or not.
- In 3.5.2 we described an inspection performed on each packet (both plain IPv6 packets and IPv6 packets embedded in SSR packets) to find whether the packet is a SIP packet, and if so, intended for the current router.

We believe any of these modifications may be optimized to improve performance, primarily the latter one, because:

- It is ran on all IPv6 packets, both plain IPv6 packets and IPv6 packets embedded in SSR
- It consists of more code that must be executed: for all packets, it must parse the IPv6 header and compare if the destination IPv6 address matches the current routers. (The subsequent parsing to see if the packet is SSR requires even more computations, but since it is only done if the packet was actually targeted to this particular router, this only affects the handling speed of these particular packets.)

5 Conclusion and future work

5.1 Conclusion

Our solution is clearly not practically usable on the Linksys MIPSEL devices. The version of Linyphi the solution it is based on is too slow on these devices to be used in more than only a very few hops under ideal conditions, and our solution lacks many basic features.

However, we have proved that our concept (implementing a SIP proxy with a DHT backend contained within the routers by extending Linyphi, to create a functional SIP service in a mesh network, without needing to make any changes on the client) could work in practice. A faster Linyphi and a more developed SIP proxy/DHT extension is all that is required to have a practically usable solution.

5.2 Lessons learned

Clearly, we should have tested the performance of Linyphi *before* starting to add functionality to it. Discovering the low performance so late was one of the major setbacks during our work.

Also, many encountered problems, and the long list of left out features which was made even longer after the completion of our work, makes us wish we had investigated and considered the solutions used in Linyphone [19] in greater depth, more specifically, the use of IGOR and a separate SIP proxy running besides Linyphi. They might have been a better solution than we first believed.

5.3 Future work

Although we have tried to leave “threads easy to pick up” in the code where future improvements could be added, we suggest no more work should be commenced until the new, rewritten, version of Linyphi is released.

It may very well be possible to adapt our extensions to the new version, however, we suggest considering using IGOR for the DHT functionality rather than our custom DHT implementation for increased performance and ease of implementation. Still, there may be some parts of our SIP code may be sufficient for the functionality needed on the routers, and could *possibly* be reused in a future implementation easier than adapting a library like SofiaSIP.

Also, before such a new implementation is *truly* useful, there needs to be more available SIP clients compatible with IPv6.

References

[1] Eng Keong Lua, Jon Crowcroft, Marcelo Pias, Ravi Sharma and Steven Lim. A Survey and Comparison of Peer-to-Peer Overlay Network Schemes. *IEEE Communications Survey and Tutorial*. 2004. http://www.cl.cam.ac.uk/teaching/2005/AdvSysTop/survey.pdf

[2] Pengfei Di, Massimiliano Marcon, and Thomas Fuhrmann. Linyphi: An IPv6-Compatible Implementation of SSR. *Third International Workshop on Hot Topics in Peer-to-Peer Systems, Rhodes Island, Greece*, 2006.

[3] ITU-T G.114: One-way transmission time. International Telecommunication Union, 2003. http://www.itu.int/rec/T-REC-G.114-200305-I/en

[4] Paul Asadoorian, Larry Pesce. Linksys WRT54G: Ultimate Hacking. Syngress Publishing.

[5] OpenWRT. http://www.openwrt.org

[6] Xiangling Wang, and C.-C. Jay Kuo. An 800 bps VQ based LPC voice recorder. The Journal of the Acoustical Society of America, 103(5):2778. 1998.

[7] ITU-T G.703: Physical/electrical characteristics of hierarchical digital interfaces. International Telecommunication Union, 2001. www: http://www.itu.int/rec/T-REC-G.703-200111-I/en

[8] SIPp. http://sipp.sourceforge.net

[9] Kent Beck and Cynthia Andres. Extreme Programming Explained: Embrace Change. Addison-Wesley, 2000. http://www.extremeprogramming.org

[10] KPhone – IPv6. http://old.iptel.org/products/kphone

[11] WIRELab Software. http://wire.cs.nthu.edu.tw/software.php

[12] Linphone: OpenSource SIP Video-phone for Linux & Windows. http://www.linphone.org/index.php/eng

[13] Sofia-SIP Library. http://sofia-sip.sourceforge.net

[14] SofSipCli. http://wiki.opensource.nokia.com/projects/SofSipCli

[15] GStreamer: open source multimedia framework. http://www.gstreamer.net

[16] SourceForge.net: KPhone. http://sourceforge.net/projects/kphone

[17] RFC 3261: SIP: Session Initiation Protocol. Internet Engineering Task Force, 2002. http://www.ietf.org/rfc/rfc3261.txt

[18] RFC 768: UDP: User Datagram Protocol. Internet Engineering Task Force, 1980. http://www.ietf.org/rfc/rfc768

[19] Johannes Eickhold (Thesis). Entwicklung einer SSR-basierten Peer-to-Peer-Telefonieanwendung für das Nokia 770. Thesis, Fakultät für Informatik, Universität Karlsruhe (TH). 2006.

[20] The distributed Video Disk Recorder: IGOR. http://i30www.ira.uka.de/p2p/videgor/igor.en.html

[21] Thomas Fuhrmann, Pengfei Di, Kendy Kutzner, and Curt Cramer. Pushing Chord into the Underlay: Scalable Routing for Hybrid MANETs. *Technical Report, Fakultät für Informatik, Universität Karlsruhe (TH).* 2006.

[22] IEEE 802: IEEE Standard for Local and Metropolitan Area Networks: Overview and Architecture. IEEE Computer Society. 2002. http://standards.ieee.org/getieee802/download/802-2001.pdf

[23] IEEE list of registered OUIs. http://standards.ieee.org/regauth/oui/oui.txt

[24] Hash functions and Block Ciphers. http://burtleburtle.net/bob/hash/index.html

[25] OpenDHT: A Publicly Accessible DHT Service. http://opendht.org

[26] Mobile Ad-hoc Networks Work group: Manet Status Pages. http://tools.ietf.org/wg/manet/

[27] F. Baker. An outsider's view of MANET (draft.baker-manet-review-01). http://w3.antd.nist.gov/wctg/manet/draft-baker-manet-review-01.txt

[28] Tadeus Uhl. Quality of Service in VoIP Communication. *AEU - International Journal of Electronics and Communications.* 58(3):178-182, 2004.

[29] Zachary A. Barnes. Is implementation of Voice over Internet Protocol (VoIP) more economical for businesses with large call centers? Master of Science Graduate Research Report, Bowie State University. 2005. http://faculty.ed.umuc.edu/~meinkej/inss690/barnes.pdf

[30] RFC 4566: SDP: Session Description Protocol. Internet Engineering Task Force, 2006. http://www.ietf.org/rfc/rfc4566.txt

[31] RFC 2396: Uniform Resource Identifiers (URI): Generic Syntax. Internet Engineering Task Force, 1998. http://www.ietf.org/rfc/rfc2396.txt

[32] David A. Bryan, Bruce B. Lowekamp, and Cullen Jennings. SOSIMPLE: A serverless, Standards-based, P2P SIP Communication System. *AAA-IEDA 2005.* 2005. http://www.cs.wm.edu/~bryan/pubs/bryan-AAA-IDEA2005.pdf

[33] Kundan Singh, and Henning Schulzrinne. Peer-to-Peer Internet Telephony using SIP. Report, Department of Computer Science, Columbia University. http://www1.cs.columbia.edu/~library/TR-repository/reports/reports-2004/cucs-044-04.pdf

[34] SIPDHT. http://sipdht.sourceforge.net

[35] Saikat Guha, Neil Daswani, and Ravi Jain. An Experimental Study of the Skype Peer-to-Peer VoIP System. 2005. http://saikat.guha.cc/pub/iptps06-skype.pdf

[36] Rüdiger Schollmeier. A Definition of Peer-to-Peer Networking for the Classification of Peer-to-Peer Architectires and Applications. *Proceedings of the First International Conference on Peer-to-Peer Computing (P2P'01) 2002.* http://csdl2.computer.org/persagen/DLAbsToc.jsp?resourcePath=/dl/proceedings/&toc=comp/proceedings/p2p/2001/1503/00/1503toc.xml&DOI=10.1109/P2P.2001.990434

[37] DD-WRT. http://www.dd-wrt.com

[38] polarcloud.com: Tomato Firmware. http://www.polarcloud.com/tomato

[39] AODV-UU: Ad-hoc On-demand Distance Vector Routing: For real world and simulation. http://core.it.uu.se/core/index.php/AODV-UU

[40] Guidelines for 64-bit Global Identifier (EUI-64) Registration Authority. IEEE Tutorial. http://standards.ieee.org/regauth/oui/tutorials/EUI64.html

[41] Sameh El-Ansary and Seif Haridi. An Overview of Structured P2P Overlay Networks. *Swedish Institute of Computer Science and Royal Institute of Technology*. 2004. http://eprints.sics.se/237/01/elansary-singlespaced.pdf

[42] Linyphi: An IPv6-Compatible Implementation of SSR. http://i30www.ira.uka.de/p2p/linyphi/

[43] Hash functions. http://www.sparknotes.com/cs/searching/hashtables/section2.rhtml

[44] P2P SIP. http://www.p2psip.org

A Source code

The source code of Linyphi 0.1 including our extensions is available on the attached CD-ROM.

B Test data

Raw test data (output from the Ping6 and SCP software we used to test the performance) are available on the attached CD-ROM.

C Useful links

We would like to supply addresses to some sites which contains information which we found useful during the development of our solution and during the writing of this paper:

A Brief Socket Tutorial
http://sage.mc.yu.edu/kbeen/teaching/networking/resources/sockets.html

Brandväggsskydd med netfilter/iptables, mer avancerade saker
http://www.lysator.liu.se/~kjell-e/tekla/linux/security/iptables/avancerad-netfilter.html

Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 Headers
http://www.ietf.org/rfc/rfc2474.txt

Error Codes - The GNU C Library
http://www.gnu.org/software/libc/manual/html_node/Error-Codes.html

Input/Output Redirection in Unix
http://www.codecoffee.com/tipsforlinux/articles2/042.html

Internet Protocol Version 6 (IPv6) Parameters

http://www.iana.org/assignments/ipv6-parameters

Internet Protocol, Version 6 (IPv6) Specification

http://www.ietf.org/rfc/rfc2460.txt

IPv6, Internet Protocol version 6

http://www.networksorcery.com/enp/protocol/ipv6.htm

IPv6 Flow Label Specification

http://www.ietf.org/rfc/rfc3697.txt

Porting IPv4 applications to IPv6

http://uw714doc.sco.com/en/SDK_netapi/sockC.PortIPv4appIPv6.html

Programming Escape Characters

http://www.wilsonmar.com/1eschars.htm

Protocol Numbers

http://www.iana.org/assignments/protocol-numbers

SIP Test Messages

http://www.cs.columbia.edu/sip/sipit/testmsg.html

Sisela

http://the.earth.li/~martin/sisela/

Sockets tutorial

http://www.linuxhowtos.org/C_C++/socket.htm

UDP, User Datagram Protocol

http://www.networksorcery.com/enp/protocol/udp.htm

Web-Based UML Sequence Diagram / UML Generator

http://www.websequencediagrams.com