TuneInNet

Flooding on the Internet Backbone

Diplomarbeit

Sebastian Dill

dill@inf.fu-berlin.de

Freie Universität Berlin

Fachbereich Mathematik und Informatik

Institut für Informatik

10. Januar 2010

Betreuer:

Prof. Dr.-Ing. Jochen Schiller

Georg Wittenburg, M.Sc.

Abstract

As its rapid growth continues and new applications and requirements arise, the technological advance of the Internet is increasingly limited by the facts and shortcomings of the existing infrastructure. Attempts to address this by incrementally extending or revising the existing standards have met with considerable difficulties, as in the case of IPv6 or Mobile IP. Although this is partly due to economical and practical reasons, the question presents itself if the fundamental technologies underlying the Internet themselves might need an overhaul. Part of the discussion of the future of the Internet thus turns towards *clean slate* designs, which envision how the Internet would look like if it were re-designed from scratch today. Purposely ignoring practical constraints, clean slate research has the liberty to challenge even the most fundamental concepts of the current design.

TuneInNet is a clean slate design developed at the Freie Universität Berlin and the Universität Zürich. It seeks to banish the intelligence from the network in a more radical manner than the Internet Protocol Suite does, thereby aiming at a backbone that consists of simple, purely optical switches and provides only raw connectivity. In this scenario, data is flooded and filtered rather than routed based on addresses. This can be construed as a move towards the radio, from which it derives its name: the sender sends data without regard to whom, whereas the receiver filters out – or tunes in to – the data that interests him. This thesis presents and evaluates a prototypical implementation of TuneInNet.

Acknowledgments

My thanks go out to the usual suspects – my parents, whose continuous and unconditional support made studies possible, as well as my brother and all of my friends who supplied me with any kind of support, advice or distraction.
Prof. Dr. Schiller provided and supervised an interesting topic that also allowed for a considerate amount of autonomy, for which I am thankful; Georg Wittenburg, M.Sc., gave any amount of input and feedback I could possibly hope for, sacrificing a lot of his time in the process and quite possibly a bit of his sanity as well.

Contents

1 Introduction

Since its inception in the 1960s, the Internet has arguably undergone the most rapid and fascinating expansion of any technology in human history. Its success has been extremely far-reaching, both figuratively and literally: figuratively far-reaching in that it transformed how our world works on both the personal and societal level, from the way we inform ourselves and socialize to the way our economies and institutions are constituted; literally far-reaching in that some 40 years after its humble beginning as the 4 host ARPANET, its millions of hosts now cover the entire globe in an ever-growing mesh of copper, fiber optics and radio waves – Figure 1 gives an artistic rendering on this, based on an actual snapshot of the Internet's structure.

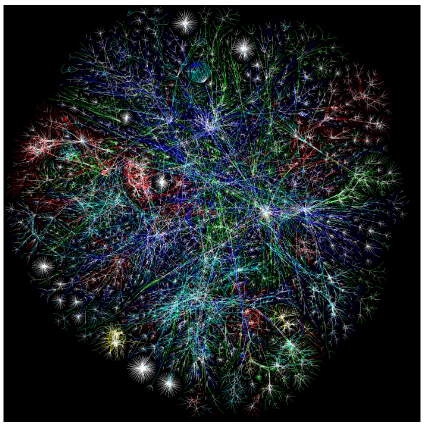


Figure 1: 2003 map of the Internet, with different regions in different colors. Red: Asia/Pacific, green: Europe/Middle East/Central Asia/Africa, blue: North America, yellow: Latin American and Caribbean, cyan: RFC1918 IP Addresses, white: unknown. Source: www.opte.org

The extraordinary speed of its success has, however, not come without cost. As neither its physical nor its functional expansion were anticipated, the limits and shortcomings of the technologies underlying the Internet are becoming increasingly apparent. Protocols designed with smaller networks in mind like IPv4 and BGP require extensions like NAT, CIDR and route aggregation in order to keep up with the growth of the network, and new applications that use the Internet in ways that it was not designed for can be cumbersome to implement.

The scientific community in general and specific groups like the IETF in particular have been industrious in keeping pace with the growing demands on the network by revising old standards and developing new ones. Prominent examples of this include Mobile IP [Per02] and the MBone [Mac94] for mobile and multicast applications respectively, DiffServ [NBBB98] for applications requiring Quality of Service, and IPv6 [DH98] as a response to multiple issues, including the address shortage and the growing demand for security built into the Internet itself.

However, the success of these measures has been limited. This partly is due to practical and economic reasons: the decentralized nature of the net makes an instantaneous global update impossible. Lacking a central administration, the adaption of a new standard depends on the compliance of the owners of the Autonomous Systems. Economic pressures exacerbate this, as without an economic incentive or explicit funding, an organization or company that owns part of the backbone infrastructure is unlikely to exchange expensive hardware. An example for this is IPv6, where the underwhelming speed of adaption can at least partly be attributed to the fact that Internet Providers see an upgrade as uneconomical. [MD07]

Another reason is that solutions may still remain suboptimal, because the desired functionality is simply too contrary to the Internet's basic design, for which Mobile IP is an example. Neither TCP nor UDP, and therefore the majority of applications, can handle a change of IP address during a session. The notion that an endpoint might want to change its network during an interaction – which is what mobility basically means – was simply not an issue when these protocols were designed. Mobile IP works around this by tunneling all traffic to a mobile device through a *home network*, so that it appears to have a fixed IP belonging to this home network. But this means significant organizational overhead, requires additional router functionality and the detour over the home network causes a larger delay and thus worse quality for multimedia applications.

With the Internet seemingly in a deadlock between the need to innovate and the inability to do so, the question emerges if the possibilities to revise and extend the existing infrastructure will at some point be exhausted. It is against this backdrop that *clean slate* ideas of the Internet are developed. Clean slate

approaches purposely ignore the practical and economical problems. Instead, they ask how an entirely new Internet would be designed from scratch with today's knowledge and requirements. This allows researchers to pursue ideas that may seem disadvantageous within the context of current technology and to challenge the fundamental dogmas of the Internet's design.
Given the multitude of the Internet's uses and aspects, the range of topics the research deals with is accordingly broad. A design of the Future Internet, as it has come to be called, must factor in many diverse problems [CDG+09]. Most of these can loosely be grouped into four categories.

- Scalability: These are the issues that perhaps are most pressing from the technical perspective. Given the size and continuous growth of the Internet, its technologies must work on a very large network and keep pace with additional growth. This generally means that cost functions must be contained. If the memory consumption or computational cost of an algorithm grow exponentially with the size of the network, it is highly unlikely to be suitable for the Internet. A specific concern is the growing hardware cost of routing devices, which might escalate in the future. [MZF06]

- Security: As its importance in the daily lives of people and society at large increases, so does their vulnerability to technical failure and malicious activity on the Internet. This means the Future Internet should supply mechanisms that ensure the privacy and integrity of data and be robust against failure of components and malicious behavior. [CDG+09]

- New application requirements: New ways to use and access the Internet have put forth new requirements. This includes the aforementioned mobility, but also multimedia applications like video streaming, which may require that the traffic meets certain requirements like an upper bound for latency. The mechanisms required for this, commonly called Quality of Service [Bla02], are in conflict with the Internet's original best effort policy, which included no guarantees whatsoever.

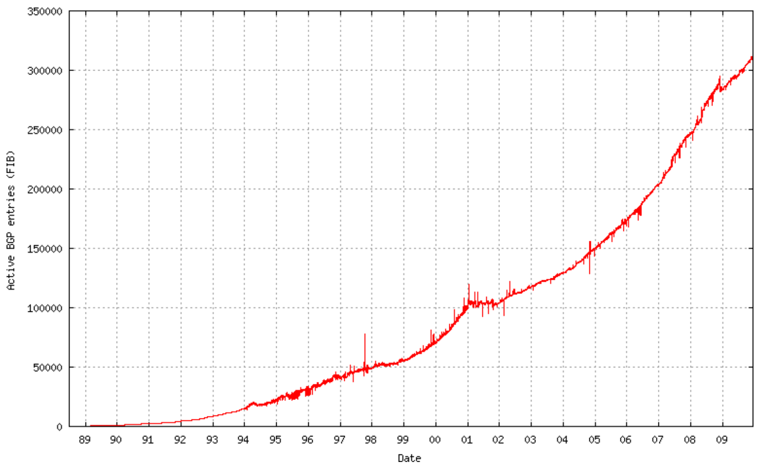


Figure 2: Active BGP entries (FIB), 1989 to 2010. Source: bgp.potaroo.net

- Social and ethical aspects: The Internet has become a major means of communication and a news medium in its own right, and this can partly be attributed to the fact that its design was principally apolitical and encouraged easy access and participation. This is generally regarded as desirable (at least the author would personally subscribe to this), and a Future Internet design should be aware of the society it helps to develop as well as its political implications.

Among scalability problems, one the most important ones is that of scalable routing. The Border Gateway Protocol (BGP), which is the *de facto* standard protocol for inter-AS routing on the Internet, has been a source of concern for a long time, because of the growth of its routing tables (pictured in Figure 2). Despite the aforementioned measures of Classless Inter-Domain Routing (CIDR) and route aggregation, the growth has not been contained in the long term. More recently, Autonomous Systems started multihoming, i.e. acquiring multiple IP ranges for their networks in order to increase their connectivity, which worsened the situation again. [YMBB05]
While possible replacements of BGP are being developed [SCE$^+$05], clean slate Internet research is at liberty to think about alternatives apart from the context of the Internet Protocol and possibly addressed-based routing in general. TuneInNet proposed to reduce, rather than to improve, the computational tasks of the network by abolishing routing in favor of a return to providing raw con-

nectivity through simple, purely optical switches. Making flooding the main mechanism of data transmission also implies giving up on the duplex channel between a pair of endpoints as the principle scenario of communication on the Internet. This may make the implementation of multimedia applications like video streaming or video conferencing easier, as these typically operate in a one-to-many or many-to-many scenario.

This thesis presents a prototypical implementation of TuneInNet and its evaluation, and it proceeds in the following steps: first, an overview over clean slate research in general and potential alternatives to address-based routing is given. Against this background, the rationale for a design that tries to radically eliminate intelligence from the network is presented. Subsequently, the prototype that was built for this thesis is introduced. Since the evaluation of the prototype required a somewhat elaborate setup, this setup is described separately. Finally, the prototype is evaluated, which includes some analytical considerations about a network topology's influence on the traffic in TuneInNet as well as the investigation into its scalability and the quality of the traffic it produces.

2 Related Work

Clean slate Future Internet research has quickly increased during the recent years and produced a large body of publications. This Section presents two of the the most important umbrella projects along with their rationale in order to establish the general context and emphasize the universal interest in clean slate approaches. This mostly relies on the overviews given in a recent paper by Anja Feldmann [Fel07] and the respective project descriptions. [Fis07] [SA09] Subsequently, a few selected research undertakings are presented, which present approaches to scalable routing that are fundamentally different from the current one. Although the details of TuneInNet switches are not part of this thesis, a very brief comment on optical switching technology is also made to show that the goal of a purely optical Internet is a reasonable one.

2.1 Future Internet Research

2.1.1 Future Internet Design (FIND)

The Future Internet Design (FIND) is an ongoing program launched by the United States National Science Foundation in 2006 with the goal "to empower the research community to design and implement a new Future Internet that builds on knowledge and wisdom about current networks, but is not constrained by the current Internet." [Fis07] It acknowledges that redesigning the Internet is not solely a technical matter and targets a broad area of research, ranging from specific mechanisms to questions concerning the social and economic dimensions of the Future Internet.

FIND funds a total of 49 different research projects and is divided into three phases, which incrementally build upon each other. The goal is to first encourage research into solutions for specific components or parts of a new network architecture, then to aggregate the findings of these projects into an overall network architecture in the second phase, and finally to implement this overall architecture.

2.1.2 Future Internet Research and Experimentation (FIRE)

The Future Internet Research and Experimentation (FIRE) program is a 2008 research program funded through the European Union's Seventh Framework Programme (FP7). It explicitly promotes clean slate research as well, stating that "there should be no boundaries for the research, but rather the freedom to address *any* emerging or radically new but promising concepts to address the fundamental limitations of the current Internet." [SA09] Similar to FIND's phases, it proposes to proceed with the Future Internet design in a bottom-up manner and puts great emphasis on early large-scale experimental testing. Given its recent creation, most of FIRE's projects are still in their early stages.

2.1.3 Rethinking Routing

The growing problems of routing are well-known, and the attempts to rethink routing for the Future Internet are plentiful. The following is a selection that exemplifies three proposed paradigm shifts, rather than specific technical matters.

One possibility is to argue that the computational intelligence should, partly or entirely, be separated from the transport protocols and possibly the routing devices themselves, so that these are controlled through external mechanisms [FBR+04]. The FIND Project "A Framework for Manageability in Future Routing Systems", for example, aims to identify "key manageability features that must be incorporated into future routing architectures, and demonstrate their benefits through development and evaluation of actual systems." [GGZ07] Similarly, the "Cooperative Management Framework for Inter-domain Routing System" [HZZL09] suggests a distributed self-organizing management framework to address the fact Autonomous Systems generally act selfish, that is, they try to maximize their own efficiency without regard to possible mutual benefits of cooperation. A classical prisoner's dilemma, this situation leads to sub-optimal global behavior, and the approach argues that a routing management framework could alleviate this fact by building "alliances" between Autonomous Systems.

A radical version of the "separation of routing and routers" stance is is presented in the "4D Clean Slate approach" [GHM+05], which distinguishes between four

types or "planes" of functionality of network protocols, the *data, discovery, dissemination* and *decision planes*, and proposes an architecture that strictly divides these planes. The discovery plane takes snapshots of the network and reports them to the decision plane, the decision plane sets the routing objectives, and the dissemination plane notifies the routing devices.
The already mentioned self-organization is the second paradigm that could be the central tool in addressing routing complexity. Self-organization attempts to create complex behavior through simple decisions that are made locally. Much of this field owes to Marco Dorigo's PhD thesis [Dor92], which introduced the Ant Colony Optimization(ACO) algorithm. Based on the observation that biological ants, through very simple, local decisions and communication with trails of pheromones, manage to solve complex path-finding problems, an algorithm was proposed which emulates this by having autonomous agents make local decisions on parts of a graph. Ant Colony Optimization has spawned a large amount of follow-up research that applies this approach to various problems, among them routing [Car04], and it may be another possible answer to scalable routing. A sample clean slate project that wants to distribute the network's intelligence is the FIRE-project ECODE (Experimental COgnitive Distributed Engine), which pursues to incorporate "distributed machine learning" into the Future Internet.
Finally, an entirely different approach to address the complexity of routing is found in "Greedy Routing on Hidden Metric Spaces as a Foundation of Scalable Routing Architectures without Topology Updates" [KCF07], a FIND projects which revisits the idea that routing on the Internet might be possible without topology updates. It argues that the Internet contains *hidden metric spaces*, i.e. distance metrics between network nodes that satisfies the triangle inequality, which is "a consequence of natural network evolution that maximizes efficiency of greedy routing on this metric space." Its first research phase attempts to prove the existence of these metrics, in order to then develop a greedy routing algorithm utilizing it. (A humorous, if not entirely correct way to paraphrase this project would be that it attempts to address the complexity of routing by arguing that it does not exist.)

2.1.4 All-Optical Internet

There is a growing consensus that the conversion of electric into optical signals is becoming a bottleneck, and that the Future Internet will consist of mainly optical devices. Daniel J. Blumenthal notes in "All-Optical Label Swapping for the Future Internet":

> "Things become very interesting when we consider that the capacity of optical fibers continues to double every 8-12 months. Today's state of the art single fiber capacity exceeds 10Tbps. Compare this increase with that of electronic processor speeds which doubles every 18 months (Moore's Law) and comes at the expense of increased chip power dissipation, we start to see that there is a potential mismatch in bandwidth handling capability between fiber-transmission systems and electronic routers." [Blu00]

That the shift towards optical technology may also mean a shift in the general networking paradigm towards broadcasting is also noted in the FIND funded project NeTS-FINDs, which notes that "optical switches are inherently a better broadcast medium than electronics and may lend to simpler architectures for multicasting, narrow-casting, multiple-access as a shared medium with contention, resolution and conferencing." [Cha07].
The idea that the optical replacements for routers should not include the entire complexity of them is also discussed in "Turing Switches - Turing machines for all optical Internet routing" [Cro03], which proposes to to focus research on optical switches with a reduced set of opcodes specifically tailored for packet switching instead of aiming at a complete von-Neumann architecture.

2.2 Position of TuneInNet

In some aspects, TuneInNet is very much in line with the current efforts to develop the Future Internet, while it stands out in others. The need for scalable routing for the Internet is widely recognized, and the decision to build a prototype early on echoes to the general consensus in FIRE and FIND that any theoretical effort towards a clean slate Internet must be accompanied by experiments. It also recognizes that the Future Internet will likely include as few electric components as possible, and the idea to make simple optical switches rather than complex optical routers may be the more economic way towards an all-optical Internet.

However, with the exception of the attempt to find hidden metrics, the presented methods of dealing with the current network's complexity all effectively attempt to refactor and outsource it rather than abandon it, like TuneInNet proposes. A possible danger of this is that the Internet's problems might simply migrate along with the complexity. The 4D approach presented acknowledges this, as it notes the danger of a "complexity apocalypse" due to the horizontal communication between its various logical planes of the network necessitate. Presenting TuneInNet's case, the next section will give an argument for the radical removal of intelligence from the network.

3 Concept and Prototype Architecture

This section gives a conceptual overview of TuneInNet and the prototype. First, an argument is made why an alternative Internet design that seeks to radically banish intelligence from the network itself is a reasonable undertaking. A description of the basic approach of TuneInNet is given, which is largely based on [CSSS07] and proposes to flood and filter data instead of routing it, thereby eliminating said intelligence hidden in the network. Subsequently, the architecture of the prototype is presented.

3.1 Rationale and Concept

3.1.1 Keeping the Intelligence out of the Network

Although not stated explicitly beforehand, one of the original design principles of the Internet – and arguably a major cause of its resounding success – was to keep the intelligence out of the network itself and instead leave all computational tasks as far as possible to the endpoints. Rather than providing specific functionality, the purpose of the Internet was regarded as providing raw connectivity. Giving a retrospective of the Internet's development, the RFC on the "Architectural Principles of the Internet" notes:

> "However, in very general terms, the community believes that the goal is connectivity, the tool is the Internet Protocol, and the intelligence is end to end rather than hidden in the network.
>
> The current exponential growth of the network seems to show that connectivity is its own reward, and is more valuable than any individual application such as mail or the World-Wide Web." [Car96]

This RFC is explicitly "in no way intended to be a formal or invariant reference model." However, if one willfully does adopt the view that the network should be as free of intelligence as possible in a more radical manner, which is what TuneInNet and this thesis do, then the current problems of the Internet can be regarded as symptoms of the supposedly little intelligence that is, in fact, inherent to the Internet Protocol Suite. By equating "connectivity" with "connectivity between pairs of endpoints" and establishing the routing of packets between endpoint pairs as task of the network itself, enough intelligence is introduced to cause considerate problems now that it has grown to a size that was not anticipated. In other words, the interpretation of the Internet's rapid growth and current problems is that the Internet's hidden intelligence was little enough to achieve its enormous success, but it needs even less intelligence in order to continue to grow.

As has already been established in Sections 1 and 2, routing is a central problem of today's Internet, and indeed a 2006 workshop report from the IETF's Network Working Group notes that Moore's Law may potentially not apply to routing technology, meaning that costs might escalate in this area. This could mean that routing would, in fact, be the limiting factor of any further development:

> "Given the uncontrolled nature of its growth rate, there is some concern about the long-term prospects for the health and cost of the routing subsystem of the Internet. The ongoing growth will force periodic technology refreshes. However, the growth rate can possibly exceed the rate that can be supported at constant cost based on the development costs seen in the router industry." [MZF06]

If address-based routing is at the core of the Internet's current problems, and if the endpoint pair-scenario as principle instance of communications hinders new applications like mobility and video streaming, then the radical paradigm of raw connectivity would suggest abolishing rather than reforming them. TuneInNet proposes exactly this. It aims at a backbone network that provides little more functionality than what is usually associated with OSI Layer 2, but would only require exceedingly simple network devices, called TuneInNet switches, which could be realized with purely optical technology. This would have the additional benefit of eliminating the need to transfer optical signals into electrical ones and back entirely.

Its name derives from the radio, an application that can be considered an example of a network utilizing raw connectivity: data is broadcasted largely indiscriminately, and it is left to the repeaters and endpoints to filter out – or to tune in to – the data that is relevant to them.

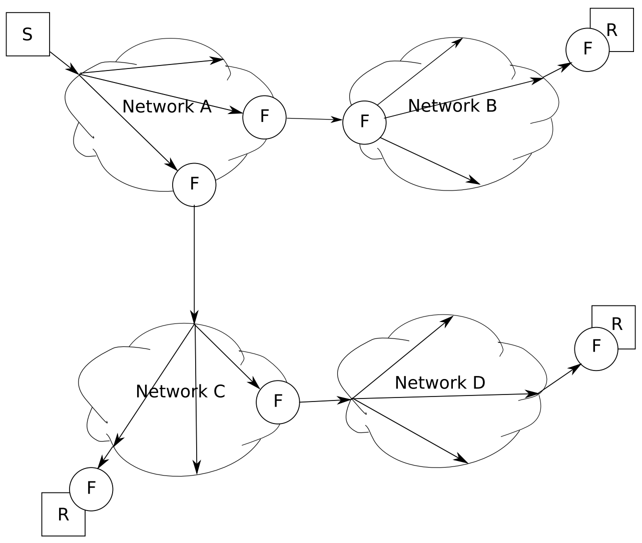


Figure 3: Schematic depiction of flooding and filtering with four networks, one sender (S), three receivers (R) and possible positions for filters (F)

3.1.2 Flooding and Filtering instead of Routing

Since the goal is to create a network with as little inherent intelligence as possible, the basic conception of TuneInNet consequently is quite simple. A sender wraps data in a TuneInNet packet, marks it as coming from him by individualizing it in some way, and sends it into the network (Autonomous System) it has access to. No topological or routing information is attached to the packets whatsoever. Any data packet that arrives in a network is flooded internally in a way that it reaches all exit points to other networks. At every exit point, it is further flooded to the respective peering network. How exactly this internal flooding is accomplished is irrelevant, although TuneInNet may possibly be used within Autonomous Systems as well. The only method to remove circling packets from the network is a Time To Live field, which is decremented every time a packet exits a network.

A network may optionally apply filters at any entry and exit point to another network, and receiving endpoints filter for relevant data as well. Figure 2 schematically depicts the flooding and possible locations of filters.

3.1.3 Total Traffic and Device Cost

The question of the amount of traffic that is generated in TuneInNet immediately presents itself. At every AS with three or more neighbors, two or more copies of a packet are created. Within densely connected parts of the network, this can quickly escalate as the copies create copies themselves, and so on. However, as the goal is to keep TuneInNet switches as simple as possible, which implies avoiding a hash of known packets, no direct measures are intended to reduce this effect at this point except for the filters. The question of TuneInNet's feasibility and economic efficiency can be translated into the question whether the benefits of having very simple, purely optical switches instead of complex routers outweighs the drawback of requiring additional bandwidth.
The evaluation in Section 4 quantifies and measures the total traffic in detail and aims to give a preliminary answer to precisely this question.

3.1.4 Exclusions: Filters and Security

This thesis' focus is the evaluation of the core idea of TuneInNet. The proposed filtering mechanisms are explicitly not addressed in detail, nor are they included in the prototype. This is done not only to limit the scope of this work, but also because they represent the point where intelligence may creep back into the design. Rather, the empirical results of this thesis shall form the basis for developing sensible filter mechanisms.
Security is a second aspect that is explicitly not covered. An obvious concern is that essentially all data becomes visible to the entire network, making way for an entire array of possible security issues. However, from the raw connectivity point of view adopted in this thesis, an argument can be made that like all other complex tasks, security should ultimately be left to the endpoints as well. The result would not be different from the current situation, where the network should generally be regarded as untrustworthy unless explicit measures are taken to secure it, e.g. by encrypting data on the application layer or by establishing a VPN. As already indicated above, a sender will generally individualize data to indicate its origin, and this may involve well-known security measures like encryption or signing.

3.2 Prototype Architecture

The following is a description of the general architecture of the prototype. It allows to build a network, in which each node represents an Autonomous System and where data packets are flooded between nodes according to the mechanism described above. As could be expected from what has been presented so far, the architecture itself contains little complexity, although its implementation and accompanying evaluation setup described in section 4 and 5 respectively do. The architecture consists of three parts:

- A TuneInNet kernel which runs on each node and floods data to other nodes
- A packet format for the communication between kernels
- An API component, which allows applications on a node to communicate via TuneInNet using the kernel

Figure 4 gives an overview of the prototype's component, and how they interact with each other and applications. The API, as its name suggest, offers the application a set of methods to use TuneInNet. The API also need its own internal interface to communicate with the kernel. The kernels each have a unique label (explained below) and communicate with each other over the network, which requires a packet format described at the end of this section.

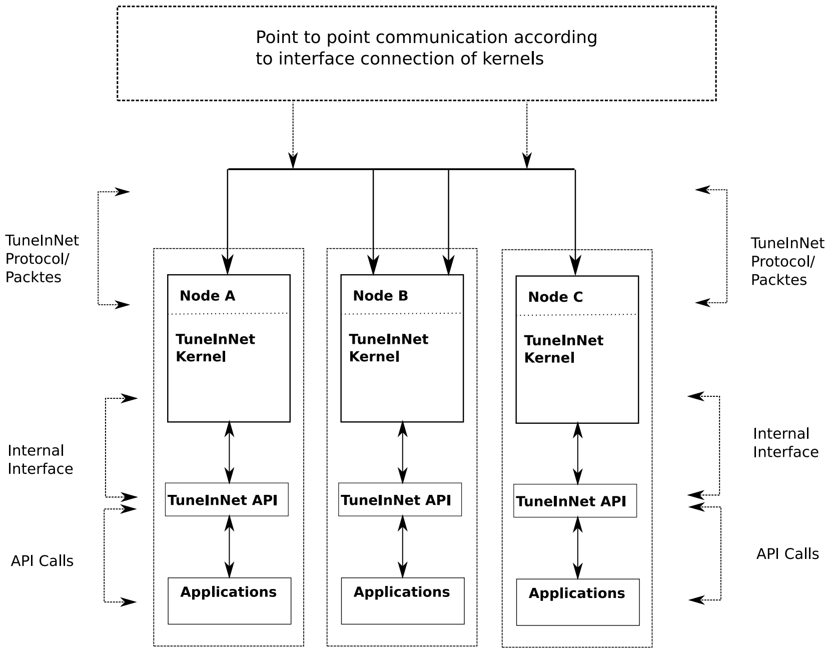


Figure 4: Overview of prototype architecture

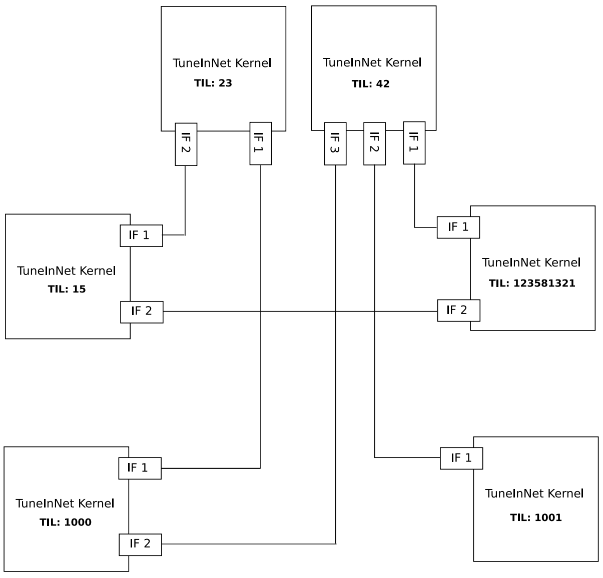


Figure 5: Five TuneInNet kernels connected via their network interfaces (IF)

3.2.1 The TuneInNet Kernel

Each TuneInNet kernel runs on a network node and controls one or more network interfaces. It has a 64 bit label (TuneInNet-label, TIL) uniquely identifying it, which must be assigned by the network administrator. Each of its interfaces is connected to an interface of another kernel, forming the topology of the network. A sample setup with five kernels can be seen in Figure 5. The task of the kernel is simply to listen on all interfaces for incoming packets, relay them to applications, and, if the Time to Live value is greater than one, reduce the Time to Live by one and flood the packet accordingly. Algorithm 1 gives the pseudo-code for this.

Algorithm 1 *TIKERNEL*

```
1: while true do
2:   receive packet $P$ from any interface $i$
3:   relay $P$ to all Applications
4:   reduce $TTL$ field of $P$ by 1
5:   if $TTL$ of $P$ is greater 0 then
6:     send $P$ through all interfaces except $i$
7:   end if
8: end while
```

3.2.2 The TuneInNet API

The minimum functionality the prototype must offer to applications are two methods for sending and receiving data. Since filtering mechanisms are lacking at this point, a simple mechanism for differentiating between data of different applications is also included by adding two 32-bit identifier to each packed, called TI-ports, which function like network-wide versions of UDP/TCP ports. The first identifies the data as belonging to (or being send to) a particular TI-port, the second identifies the TI-port the application will use to identify a possible reply (or listen on). In keeping with the radio metaphor, one might conceivably also call TI-ports *channels*, but since the implementation maps these to UDP ports, this would be somewhat excessive in this context.

API Call	**Semantic**
TI-SEND(sport, dport, data)	send (flood) **data** into the network marked with identifiers **dport** and optional reply port **sport**
TI-RECEIVE(dport)	receive data from the network marked with identifier **dport**

Table 1: TuneInNet API methods

The API consequently includes but two methods, which are listed in Table 1. **TI-SEND** floods data to a TI-port, and optionally marks the broadcast with a port it expects a reply on. **TI-RECV** receives data on a TI-port and additionally returns the TI-port the originator of the data listens on.

3.2.3 The Packet Format

As mentioned above, the kernels flood data through their network interfaces to each other, which requires a data packet format. From the considerations so far derives a packet format that can be seen in Figure 6 and consists of seven fields:

- **Sender Label:** The 64 bit label of the kernel who originally send the packet.
- **Sender TI-Port:** The TI-port on which the sending application may expect a reply
- **Destination Label:** A 64 bit label, identifying the intended receiving node(s) of the packet. At this point, this is optional, because every node will receive every packet.
- **Destination TI-Port:** The TI-port for which the data is meant
- **TTL:** The Time To Live of the packet
- **Payload Length:** The size of the data in bytes
- **Data:** The application's payload or the next layer's header

As stated initially, none of the prototype architecture's components contains any significant amount of complexity. The algorithm the kernels use is exceedingly simple, as are the API methods' signatures and the packet format.

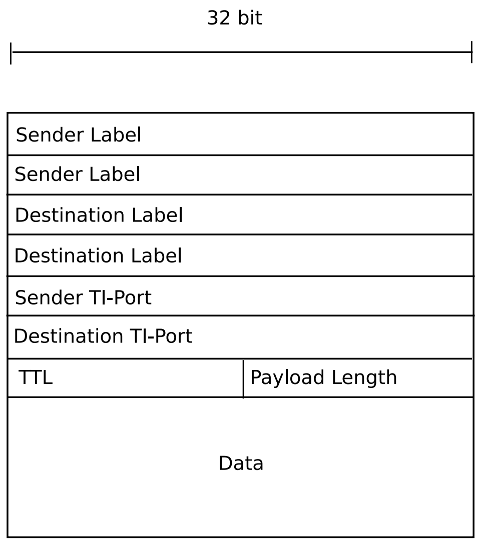


Figure 6: TuneInNet packet format

Designing a corresponding network device would be an easy task compared with that of designing a router. Implementing the architecture on top of the IP Suite, however, requires some less obvious measures, which are described in the following section.

4 Implementation

The prototype has been implemented on top of the Internet Protocol Suite in order to evaluate TuneInNet using existing hard- and software. This choice translates the architecture components of section 3 as follows: the TuneInNet kernel is implemented as a server daemon, which floods data through point-to-point connections to other kernels. The TuneInNet API component takes the form of a shared library, which offers the UDP subset of the Unix socket API by intercepting system calls. The implementation was done on GNU/Linux in plain C; C being the obvious choice for system and network programming on a POSIX-compliant OS.

4.1 The Server Daemon

The server daemon mainly implements the algorithm given in section 3 with sockets representing network interfaces. It receives packets on one socket and copies them to a number of other sockets, if the packets have not outlived their Time To Live. Both TCP and UDP are supported as transport protocols; the reason for implementing both is explained by the results of preliminary tests described in the next section.

A second explanation that must be delayed is how the daemons find each other. Conceptually, they have no topological knowledge whatsoever and simply copy data between network interfaces. However, the network interfaces are implemented as network sockets, and network sockets effectively need some sort of peer. A stringent implementation would theoretically be possible by finding out the system's physical network interfaces and sending data to their respective broadcast addresses using UDP, but this would be very impractical and extremely unwieldy, because for each test setup, the hardware would have to be rewired.

In practice, the codebase of the prototype includes a virtual network layer for the purpose of evaluating larger networks, which implicitly solves this problem and is described in section 5 as well.

Independent of what is chosen as transport protocol, the daemon does listen on a UDP port for control messages from applications. It also maintains a statistic of how many packets it receives and sends as well as a simple data structure that keeps track of which TI-ports applications are interested in. The daemon could simply relay all data to all applications, but this would increase the CPU load, because every application would process every packet. This way, packets are pre-processed in the daemon and relayed selectively.

```
struct subscription
{
    int TIport;
    int COMport;
};
```

The TIport field contains the TI-port the application wants to receive data for; the COMport field is an actual UDP socket where the data will be sent. It should be noted that the fact that the server sends the data to the shared library using UDP is coincidental. It was chosen as means for inter process-communication, because the need to relay packets between two components suggests the use of network sockets , but it could also have been implemented using signals. An application may listen on the TI-port 53, thinking that it is occupying the host's well-known DNS port, but it will actually receive data from TuneInNet, which the shared library receives through an arbitrary UDP port.
The daemon also must keep a hash of checksums over the data packets it receives. UDP applications do not expect duplicates to be handed to them, because these are usually detected and dropped by the operating system. However, multiple packets with the same payload legitimately arrive at a TuneInNet kernel, so it must similarly ensure that only one copy arrives at every application. This is done by assigning each packet a hash checksum, which the kernel checks against a list of known values.

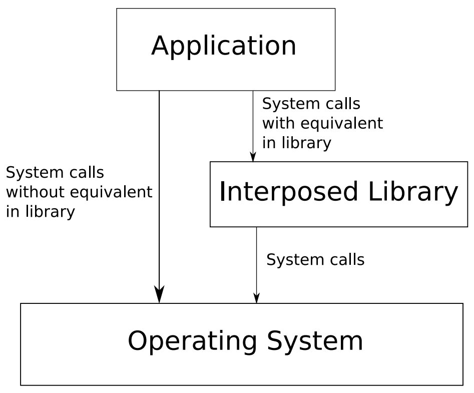

Application
System calls with equivalent in library
System calls without equivalent in library
Interposed Library
System calls
Operating System

Figure 7: Library Interposition

4.2 The Shared Library

4.2.1 Library Interposition

The TuneInNet API is provided through a shared library that intercepts system calls. This process, schematically depicted in Figure 7, is called library interposition. It advises the dynamic linker to try to resolve symbols in the interposed user space library before looking for them in the system's libraries. Thus, if an application attempts to make a system call and the library contains a method with a signature that is identical to that of the system call, this user space method will be called rather than the system call the application intended. The interposed library may still chose to access the original system calls.

4.2.2 TuneInNet Sockets and Changed System Call Semantics

In order for existing application binaries to work, the semantics of the system calls must be preserved in a way that calls with valid parameters result in valid return values and behavior. When using UDP, applications expect to create a socket (which is a specific type of file descriptor), and then perform various operations on it, like binding it to an address, receiving data from it or closing it. To the program, it must seem like the system calls of the subsystem behave in a manner consistent with these expectations.

This is achieved as follows: when a program initially attempts to create a UDP socket, the shared library does return an integer identifier called a TI-socket. This identifier is, in fact, a file descriptor that the library created, so a file descriptor created elsewhere in the same process is not confused with it. The TI-socket also serves as a key in a hash map which maintains a data structure for remembering which addresses and ports the application assumes to be its peers.

Whenever a system call is made that operates on a file descriptor, the library checks if the identifier is a TI-socket. If this is the case, TuneInNet-functionality is executed. If an application provides an address for the first time, the daemon is informed that the application wants to tune in into the respective TI-port. This can happen explicitly through **bind()** or **connect()** call, but it is also the case for the **sendto()** and **recvfrom()** calls, as a UDP sockets may rely on the operating system to provide arbitrary free ports. In the latter case, the library may have to assign the application an arbitrary TI-port where it may receive data. If the application wants to send data with either **send()** or **sendto()**, a TuneInNet data packets is created and send to the daemon for flooding. If it attempts to receive data with either **recv()** or **recvfrom()**, the library checks if the daemon has send any data packets to its internal UDP port, removes the packet's TuneInNet header and returns the payload. If the identifier is not a TI-socket, control is handed over to the original system call. The library must also allow DNS related calls to function as usual in order to still allow applications to resolve names. Figure 8 depicts the workflow for this.

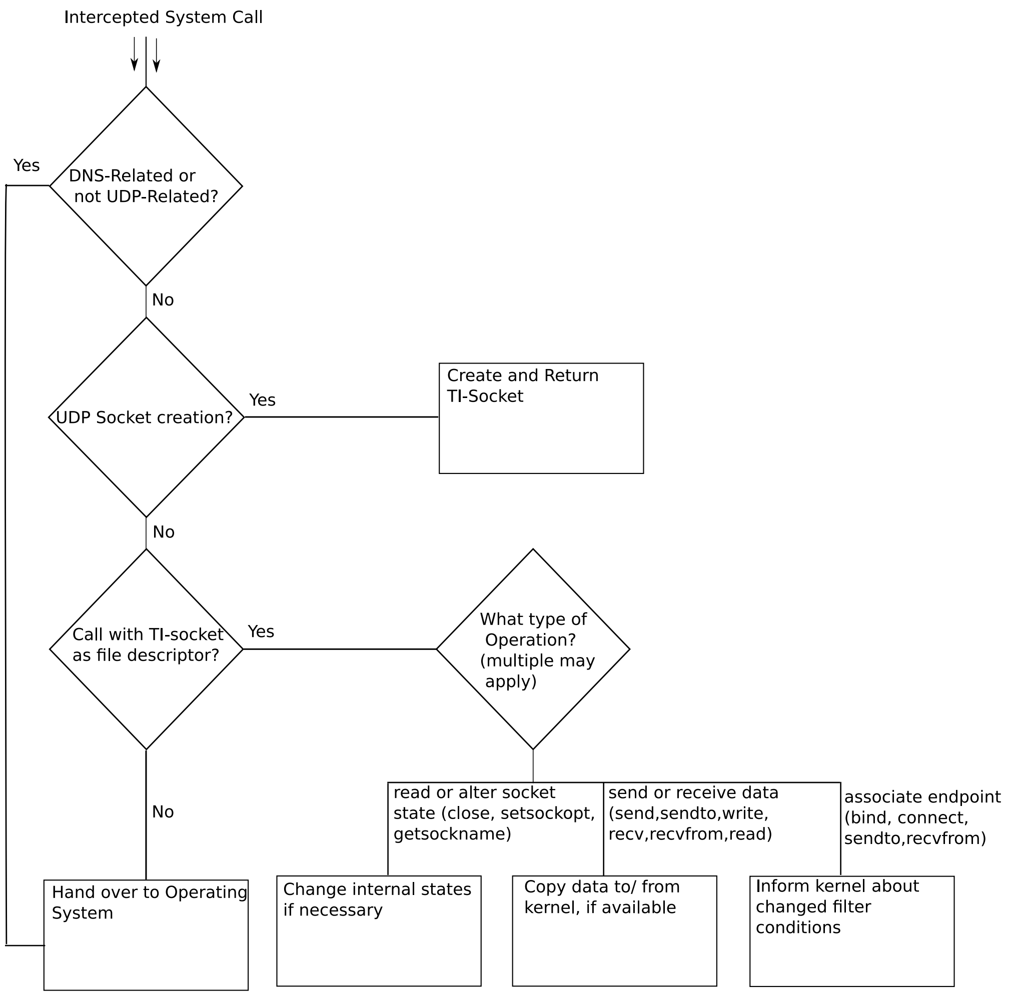

Intercepted System Call
DNS-Related or not UDP-Related?
Yes
No
UDP Socket creation?
Yes
Create and Return TI-Socket
No
Call with TI-socket as file descriptor?
Yes
What type of Operation? (multiple may apply)
No
read or alter socket state (close, setsockopt, getsockname)
send or receive data (send,sendto,write, recv,recvfrom,read)
associate endpoint (bind, connect, sendto,recvfrom)
Hand over to Operating System
Change internal states if necessary
Copy data to/ from kernel, if available
Inform kernel about changed filter conditions

Figure 8: Workflow of the Shared Library

Table 2 lists the POSIX calls that are intercepted (with greatly simplified semantic and parameter list for clarity) and how they are mapped to the TuneInNet API-methods listen in section 2. The change in semantics of the system calls is more visible to a user than to the application, because the main difference is that the ports an application presumes it listens on and sends to are not unique to a host's IP address. The resulting behavior is essentially an IP network, in which UDP ports are network-wide rather than host-specific.

For example, using the VLC media player over TuneInNet, it is possible to stream a video to every node in the network at once by using what appears to the application to be a regular UDP unicast socket. This works well, since VLC expects no answers from the client. It is also possible to run UDP applications which perform duplex communication, as long as the instances run on different ports.

Another major difference is that no errors are reported to the application, because the measures the application would then take would not be meaningful in most cases. For example, if the library fails to contact the daemon, the application could do nothing to diagnose or remove the problem. If it encounters an error, the shared library aborts the program.

System Call (simplified)	**Original Semantic (simplified)**	**New Semantic for TI-Sockets**
socket()	create an unbound socket and return file descriptor	if attempt to create UDP socket create a TI-socket and return it, otherwise call original system call
sendto (data,IP:port)	send **data** to **IP:port**	**TI-SEND(sport,dport,data)** where **dport** = **port**
connect (IP:port)	determine **IP:port** where data will be send to with subsequent send() calls	determine **port** as **dport** for subsequent **TI-SEND** calls
send (data)	send **data** to host previously connected to with connect()	**TI-SEND(sport,dport,data)** where **dport** is specified with previous connect() call
bind (IP:port)	assign name **IP:port** to socket,if possible	determine **port** as **rport** for subsequent **TI-RECV** and **sport** for subsequent **TI-SEND** calls
recv()	return data pending on bound socket	**TI-RECV(rport)**, where **rport** is specified with previous bind() call
recvfrom()	return data pending on bound socket and originating IP address/ port	**TI-RECV(rport**) where **rport** is specified with previous bind() call additionally, return **sport** from received data and an arbitrary IP address
getsocket-name()	return name (IP:Port) of socket	return TI-Port assigned by library
write(data)	as send()	as send()
read()	as read()	as read()

Table 2: System calls mapped to TuneInNet API methods by the shared library

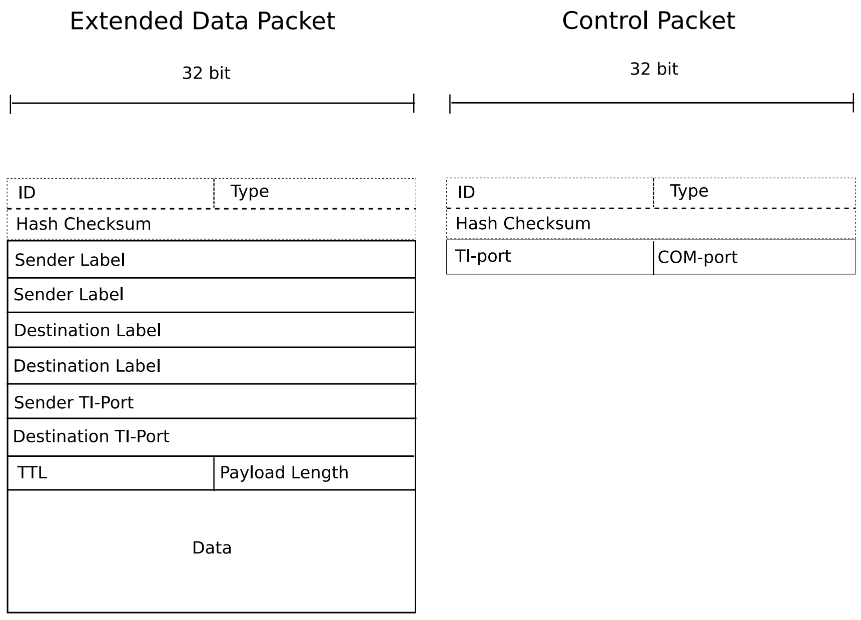


Figure 9: Extended TuneInNet Packet Format and Control Packets

4.3 The Extended Packet Format and Control Packets

The implementation on top of the IP suite necessitates several additional fields in the TuneInNet packets. The packet format must also include control packets and distinguish them from data packets, because the shared library and the daemon communicate with each other over UDP sockets. These fields are:

- **ID:** Two bytes to distinguish TuneInNet packets from stray UDP packets that may randomly be sent to a port
- **Hash Checksum:** As explained above, UDP applications do not expect duplicates to be handed to them, and the daemon needs a checksum to to identify packets with semantically identical payload
- **Type:** If UDP is chosen as transport protocol, the server daemon must be able to distinguish between data packets from other daemons and control packets from applications. There are a total of thee types of packet:

  - *Data:* a TuneInNet Data packet
  - *Tune in:* an application informing the daemon that it is ready to receive data on a specific TI-port
  - *Tune out:* an application informing the daemon that it no longer wants to receive data on a TI-port

The control packet's two diverging fields have the following function:

- **TI-port:** The TI-port the application wants to tune in to / tune out of
- **Com-port:** The regular UDP socket the shared library created for communication with the daemon

In sum, the implementation mirrors the architecture described in section 3 closely, albeit with a few additions. The server daemon is an obvious implementation of the kernel, and the extended packet format is necessary since the implementation works on top of IP. When looking at the two tables listing the API methods in this and the last section, it may appear that the implementation's socket API is rather cumbersome when compared to the two simple methods described in the architecture. However, the process of making an application do something else than it wants to generally involves very specific measures, and the result contains some unavoidably complexity. It should be noted that the implementation's API is created in order to evaluate the prototype, not to provide an interface to developers.

5 Evaluation Setup and Auxiliary Implementation

This section describes the setup that was used during the evaluation of the prototype. This comprises the hardware and external software that was used, preliminary measurements to determine limits of the hardware, and additional components that were implemented but are, conceptually, not part of TuneInNet itself.

5.1 The Virtual Network Layer

As already noted in the preceding section, a virtual network layer has been added to the prototype that allows to run several network nodes on one physical host. The fact that the network interfaces are implemented as sockets makes this relatively easy by defining peering server daemons in an XML configuration file, which is shared by all server daemons. This allows to test networks that are larger or topologically different from the one that is physically available. Figure 10 indicates that, conceptually, the kernel still knows nothing of the topology – it is not relevant to its flooding algorithm how the network interfaces are created. Figure 11 shows how the complete implementation effectively works with the virtual network layer. The server daemon reads an XML configuration file upon startup that allows it to create appropriate sockets. The daemons can then communicate with each other over the sockets using the TuneInNet packet format. The server daemon's TuneInNet labels, having no topological significance, are still chosen arbitrarily and have no relation to the IP address of the physical host they run on or to each other.

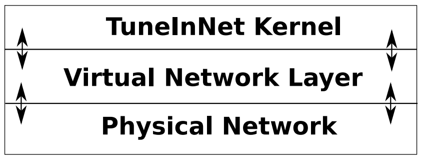


Figure 10: The Virtual Network Layer, separating the Kernel from the physical network

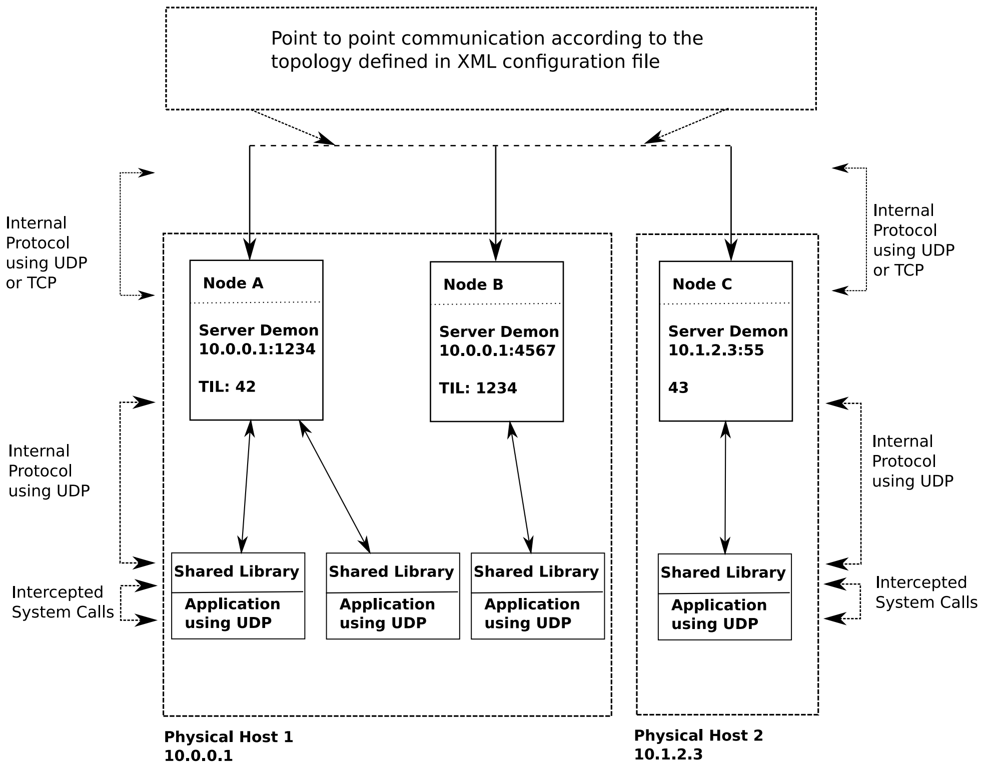

Point to point communication according to the topology defined in XML configuration file
Internal Protocol using UDP or TCP
Internal Protocol using UDP
Intercepted System Calls
Node A
Server Demon 10.0.0.1:1234
TIL: 42
Node B
Server Demon 10.0.0.1:4567
TIL: 1234
Node C
Server Demon 10.1.2.3:55
43
Shared Library
Application using UDP
Physical Host 1 10.0.0.1
Physical Host 2 10.1.2.3

Figure 11: Schematic evaluation setup with three TuneInNet nodes A,B and C running on two physical hosts and serving four applications

5.2 Bandwidth Restriction with Leaky Buckets

Given that the application using TuneInNet will take up more bandwidth than without using it and that any physical link serves for several virtual network links, the physical limits of the network are quickly reached if the sever daemons are allowed to use the physical bandwidth without restraint. It is thus necessary to artificially limit the the bandwidth for the virtual network links. This is done by using the Leaky Bucket Algorithm, also known as Generic Cell Rate Algorithm in the context of ATM. [Bla02]

The algorithm is, metaphorically, similar to a bucket that has holes at the bottom. Water pours out at a constant rate, no matter how fast it is added to the top, and if additional water is poured in while the bucket is full, it flows over and never reaches its destination. Transferred to packets, this means that packets which are supposed to be send over a network link are placed in a FIFO queue with limited capacity, and then only send out at a constant rate.

Algorithm 2 *LEAKYBUCKET*

```
1: Let $\Delta$ be the time interval and $M$ the maximum bytes to be sent out per interval
2: Let $Q$ be a queue of packets and $HEAD(Q)$ the next packet
3: Let $PAY(x)$ denote the payload size of a packet $x$ in bytes
4: Let $SENTBYTES$ be an integer
5: $SENTBYTES := 0$
6: while true do
7:    take packet $P$ meant for network link, if available. Place it in $Q$, if $Q$ is not full. Otherwise, drop it.
8:    while $Q$ not empty and $PAY(HEAD(Q)) + SENTBYTES < M$ do
9:       $SENTBYTES := SENTBYTES + PAY(HEAD(Q))$
10:      remove packet $HEAD(Q)$ and send it over the link
11:   end while
12:   if $\Delta$ has passed then
13:      $SENTBYTES := 0$
14:   end if
15: end while
```

5.3 Hardware and Software Setup

The available hardware consisted of twenty physical hosts, ten of which were equipped with 2.8GhZ Quad-Core Pentium processors and 8GB of physical RAM. The other ten were 2GhZ dual-core Pentiums with 2GB of RAM. They were running x86_64 GNU/Linux and i686 GNU/Linux respectively and were not available exclusively during the evaluation, i.e. varying CPU load and background network traffic during the evaluation was to be expected. The groups of ten older and ten newer hosts are all connected with 1 GigE switches.
The main program used was iperf. [IPE] Iperf, a well-known network tool, can be run over UDP, which means that it can be used with the TuneInNet shared library to produce streams of specific bit rates or measure jitter. The VLC media player [VLC] was also used for the use case of video streaming.

5.4 Hardware Limits

5.4.1 Transport Protocol Choice Rationale

Throughout the evaluation, TCP was used as transport protocol between the server daemons. Initially, UDP might seem to be a more suitable choice as transport protocol than TCP. Loss of single packets would not be very significant, because data is flooded with high redundancy, and the prototype also does not offer stateful connections to applications. However, preliminary testing showed that this makes the Linux kernels' UDP buffers the limiting resource, which leads to a considerable amount of packet loss. Figure 12 shows a comparison of the behavior of TCP and UDP during a typical test run. Streams of data with Constant Bit Rates ranging from 1 to 5 kBit/s were sent through TuneInNet, and the PDR was measured at every node. The Figure plots the average PDR for all nodes as well as the worst node's value in each case.
As can clearly be seen, TCP continues to run fine where single nodes in the UDP case already break down. The reason the UDP worst case drops so sharp while the UDP average PDR remains fairly good is that the first node to fail is generally a very central node, and this takes significant load of the rest of the network. The confidence intervals in the UDP worst case also indicate the high volatility of this process.

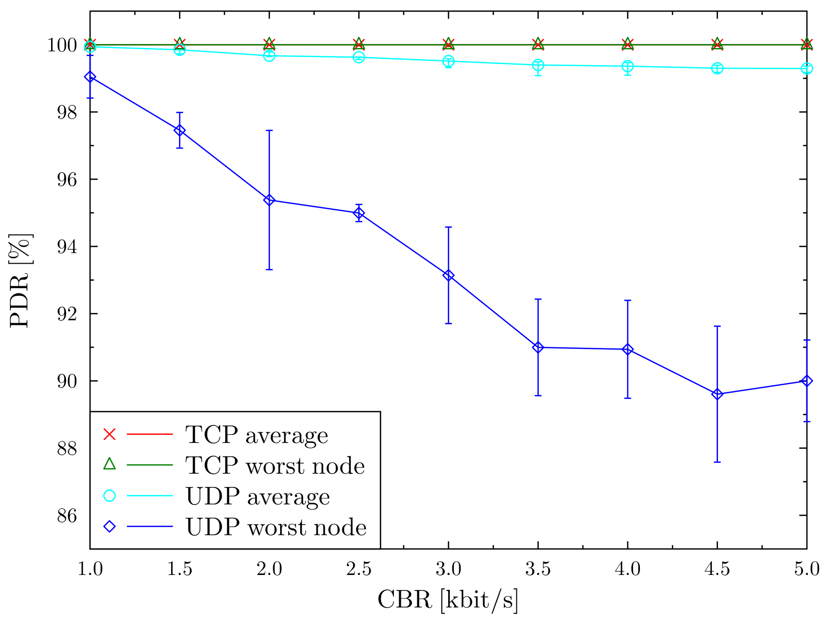


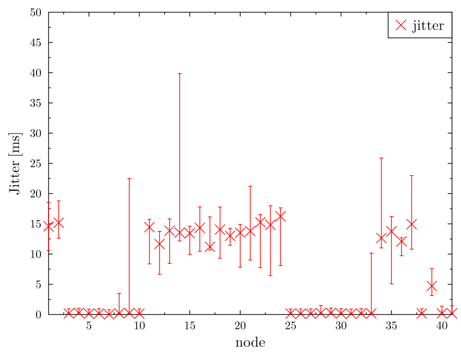
Figure 12: PDR for TCP and UDP with same test setup

Since these effects are highly undesirable – any significant amount of packet loss should only occur in a controlled manner by the prototype, not through the physical system – TCP was exclusively used during the evaluation. The limiting factor in this case is the processing time the server daemons need. The newer hardware supports a maximum of about five to six server daemons, the older hardware about three to four, which makes a network of about 100 nodes the largest observable case.

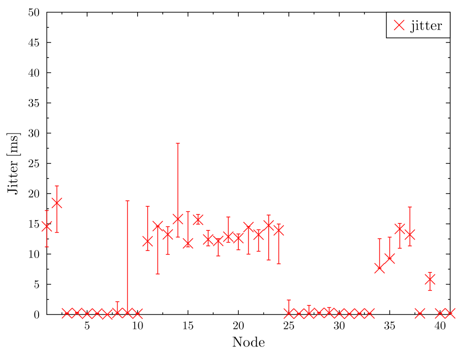
5.4.2 Hardware Effects on Time-Critical Measurements

Figure 13 shows two sample jitter measurements that were conducted with iperf in a network of 40 nodes. A stream of data was sent from a node into a network, once with a CBR of $500kb/s$ and once with $1000kb/s$. Jitter was measured on every node using iperf.

There is a clear division into two groups of nodes: one with a jitter of about 15ms to 20ms, the other with values well below 1ms. This division does, in fact, mirror the difference in the two types of underlying hardware. Since both newer and older hosts are connected with the same Ethernet hub and measuring jitter without TuneInNet gives similar values between old and new hosts, it appears



(a) CBR = 500 kb/s



(b) CBR = 1000 kb/s

Figure 13: Initial jitter measurement

that a higher CPU speed makes a significant difference in the speed at which the prototype handles packets. This means that for time-critical measurements of this granularity, the abstraction the prototype offers from the physical properties of the hardware is not enough, and to remove this effect, far more sophisticated traffic shaping would be necessary.

5.5 Auxiliary Components

5.5.1 The Reference Multicast Implementation

In order to have a frame of reference for jitter measurements despite the problems just described, the prototype supports routing its packets via multicast as an alternative to flooding.

For this purpose, a Steiner tree is constructed over the TuneInNet network graph. A Steiner tree is a minimum spanning tree over a subset of designated vertices and as few additional vertices as necessary, and is a common method to chose the path for multicasting data to a designated set of receiving nodes. [Win87] Since this is done only once at start-up and the performance of it is therefore not very relevant, a common heuristic algorithm that is easy to implement has chosen, which can be broken down into shortest path calculations and the construction of minimum spanning trees. [Meh88] [KMB81]

Algorithm 3 *Steiner − H*

```
1: Let $G = (V, E, d)$ be a connected, undirected distance graph, and $S \subseteq V$ the set of vertices for which a Steiner tree is desired.
2: Construct the complete distance graph $G_1 = (V_1, E_1, d_1)$, where $V_1 = S$ and for every $(v_i, v_j) \in E_1$, $d(v_i, v_j)$ is equal to the distance of a shortest path from $v_1$ to $v_j$ in $G$.
3: Find a minimum spanning tree $G_2$ of $G_1$.
4: Construct a subgraph $G_3$ of $G$ by replacing each edge in $G_2$ by its corresponding shortest path in $G$. (If there are several shortest paths, pick an arbitrary one.)
5: Find a minimum spanning tree $G_3$ of $G_3$.
6: Construct a Steiner tree $G_5$ from $G_4$ by deleting edges in $G_4$, if necessary, so that no leaves in $G_5$ are Steiner vertices.
```

5.5.2 The Topology Generator

Since TuneInNet should be tested for a large amount of topologies, it is reasonable to automatize the process of creating topologies. Since existing generators that produce realistic Internet-like topologies such as inet [Jar02] operate at a far larger scale than the prototype supports, a special generator was written specifically for the TuneInNet prototype. The generator reads a skeleton XML configuration file which contains global configuration variables and a list of the available hardware resources, and then then produces an XML configuration file for the prototype. Possible parameters are:

- The number of Nodes $|V|$
- The number of Edges $|E|$
- The minimum degree of nodes
- The maximum degree of nodes
- The Minimum Shortest Path Length
- The Maximum Shortest Path Length
- The number of nodes to mark as multicast recipients

(The relevance of the shortest path lengths will be explained in the evaluation section.) The generator employs a purely random approach so that no specific class of topology is favored:

Algorithm 4 *GENERATE*

```
1: Start with an Empty set of edges $E$
2: while $|E|$ is below the the specified amount of edges the graph should have do
3:     Randomly select two nodes whose degree is below the maximum degree. Add an edge between these nodes.
4: end while
5: Check if the graph is connected and meets the boundaries.
```

The resulting topology is written as a TuneInNet XML configuration file.

5.5.3 The XML Configuration Format

The entire setup is configured with the XML just mentioned, and which is generally created by the topology generator. It has the following fields:

- **Mode:** Valid values are TUNEIN and MULTICAST, determines if the data is flooded or routed it with multicast
- **Protocol:** Valid values are UDP and TCP, determines the transport protocol used for communication between daemons
- **TTL:** an integer setting the initial TTL of packets
- **LBBPS:** The bandwidth of the leaky buckets in bits per second
- **LBSize:** The size of the leaky buckets' queues
- **Node:** A node specifies a server daemon. Subfields are:
  - *Label:* The TuneInNet label, uniquely identifying the node
  - *Name:* A unique name, for making nodes easier to distinguish
  - *IP:* The IP address the node runs on
  - *Port:* The port the daemon listens on

  - *Multicast:* Either "y" or "n", indicating if this node is to be connected with the Steiner tree in the case of multicast

- **Connection:** Specifies that two nodes are connected. Subfields are:
  - *node1:* Name of first node of connection
  - *node2:* Name of second node of connection

A sample configuration file is attached in appendix A.

5.5.4 Evaluation Automatization

In order to produce a single data points for meaningful measurements, there are generally many steps are necessary:

1. Start the Topology Generator and produce a network topology according to the script's parameter. In subsequent runs, this may be done with incremental changes in the number of nodes or connections, etc.
2. Read the resulting XML file and start the server daemons accordingly
3. Make sure the server daemons are running
4. Start applications that are supposed to receive data, if any
5. Start the application that sends data
6. Wait for the sending application to finish
7. Shut the system down
8. Read and process the resulting logfiles

Since this is a laborious process, it was automatized with a set of scripts. Figure 14 gives an overview of a typical run of a measurement performed by an evaluation script.

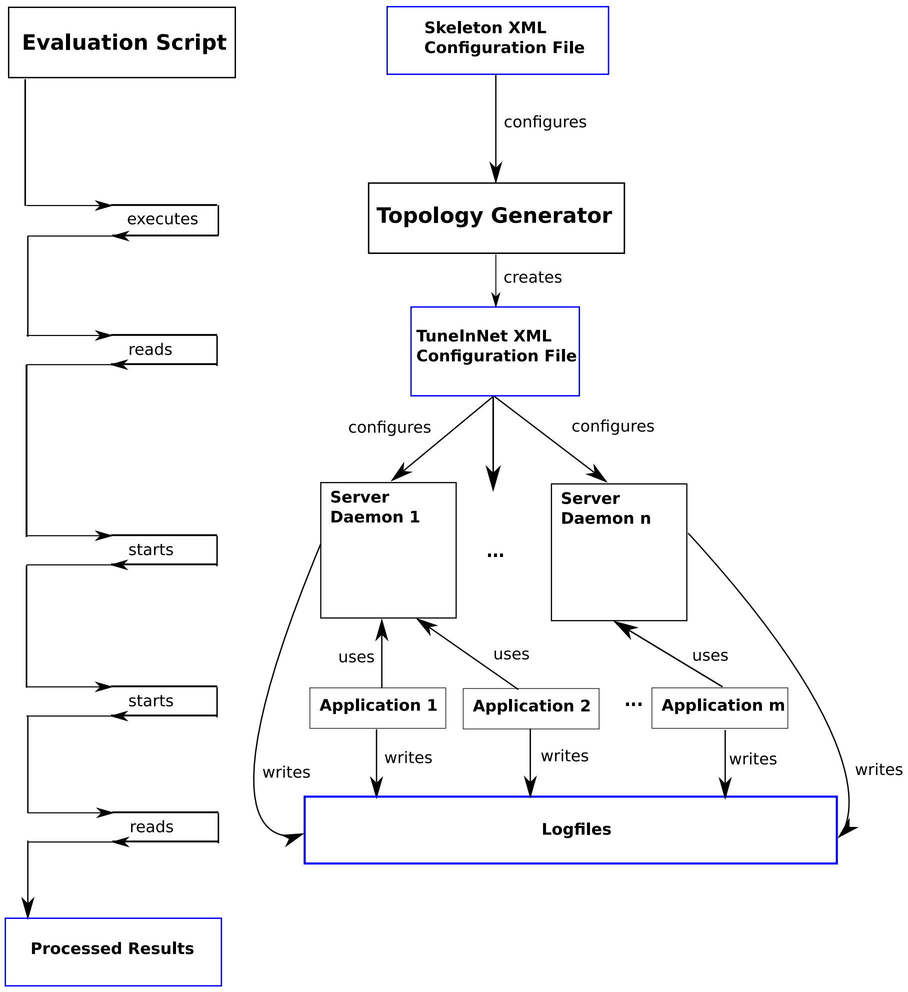


Figure 14: Workflow of an Evaluation Script

6 Evaluation

The evaluation of the prototype proceeds in four steps. As very first test, the use case of video streaming is presented to demonstrate that prototype does work with existing applications and is, in principle, suitable for multimedia applications. Second, a metric on network topologies is introduced and a few analytical observations are made about the behavior of TuneInNet under various topologies. The metric will help in choosing topologies for the subsequent tests, but it also doubles as a tool to further test the implementation. Third, jitter is measured as a quality of the traffic that is of interest to streaming applications. Fourth and finally, the scalability of the prototype is investigated.

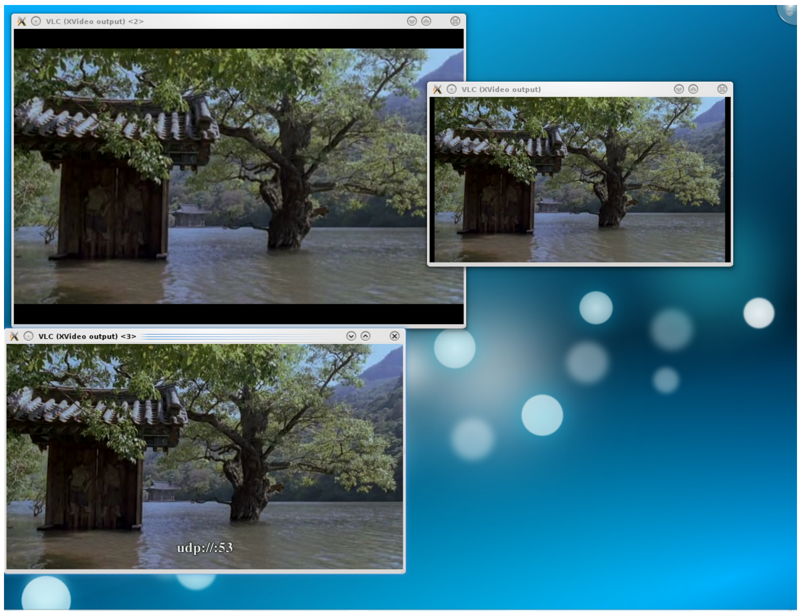


Figure 15: Screenshot - three VLC instances receiving a video stream with TuneInNet

6.1 Use Case: Video Streaming

As proof that the TuneInNet prototype can indeed be used for streaming applications, the VLC media player was the first application to be tested. VLC allows to stream videos over UDP and can thus be run with the TuneInNet shared library. Figure 15 shows a screenshot of three instances of VLC receiving the same stream, which a fourth VLC instance streams into the TuneInNet network. The third application has just tuned in and shows on its on-screen display that it apparently receives the stream on the well-known DNS port 53, even though it is running in user space. It is, of course, receiving data on the network's TI-port 53 instead, like the other three instances, even though they all assume that they are receiving a unicast stream directed specifically at them.

6.2 The Total Traffic Modifier Φ

As one will expect the total network traffic in TuneInNet to be rather significant when compared to classical routing approaches, an obvious part of the evaluation of TuneInNet is to investigate the traffic it produces in different network topologies. For this, a metric Φ, the Total Traffic Modifier, is defined, which requires some semi-formal definitions. For the purposes of this thesis, a topology is essentially a network graph grouped with the length of the longest of all shortest path lengths in the graph:

- A **topology** $T = (E, V, \vartheta)$ is a simple, connected, undirected, unweighted graph, with V being the set of vertices, $E \in V \times V$ the set of edges and $\vartheta \in \mathbb{N}$. Since the context is networking rather than graph theory, "vertex" will be used interchangeably with "node" and "edge" with "connection" respectively. Where not mentioned otherwise, ϑ is assumed to be the the length of the longest all shortest paths in (E, V) and also called **maximum shortest path length (MSPL).**

The MSPL is generally the reasonable initial value for the packets' Time to Live field, because every node must be able to reach each other. Throughout the evaluation, the initial TTL field was set to the MSPL of the topology.

Based on this definition of a topology, as a first step towards a metric for total traffic, it is possible to calculate the exact amount of times a packet will be sent through any network link in the entire network when sent from a specific start node. Packets with (semantically) identical payload are called *copies* of each other rather than duplicates, since multiple packets with the same payload will legitimately be sent through the same network link in TuneInNet.

- Given a network topology $T = (E, V, \vartheta)$ and a start node $v \in V$ from which a packet is originally sent, the **amount of copies** that will be created of a packet in the network is denoted by $P_c(T, v)$. This value can be calculated as follows:

  Let $n = |V|$. Let $A = [a_{x,y}]_{n \times n}$ be the complete adjacency matrix of (E, V) and $i_v \in \mathbb{N}$ be the index of v in A. Let $B^{(0)} = [b^{(0)}_{x,y}]_{n \times n}$ with $b^{(0)}_{x,y} = 1$ if $x = y = i_v$ and $b^{(0)}_{x,y} = 0$ otherwise. $B^{(k)}[b^{(k)}_{x,y}]_{n \times n}$ is defined recursively for $k > 1$:

$$b^{(k)}_{x,y} = \sum_{z \in \{1..n\} \setminus \{x\}} b^{k-1}_{y,z} \cdot a_{x,y}$$

  Intuitively, $b^{(k)}_{x,y}$ contains the amount of packets that node x would receive from node y with a TTL of k, making the calculation of $P_c(T, v)$ for a topology T straightforward by simpling adding up all packets every node receives:

$$P_c(T, v) = \sum_{i=1}^{\vartheta} \sum_{x=1}^{n} \sum_{y=1}^{n} c^{(i)}_{x,y}$$

As a metric for traffic overhead, this is however not entirely satisfactory, because the value of $P_c(T, v)$ still depends on the entry node v. In order to get a measure of how much traffic to expect in a given topology independent from where data originates, the natural next step is to calculate the average of $P_c(T, v)$ over all entry points $v \in V$.

- The **Total Traffic Modifier** $\Phi(T)$ of a topology $T = (E, V, \vartheta)$ is therefore defined as:

$$\Phi(T) = \frac{1}{|V|} \cdot \sum_{v \in V} P_c(T, v) \tag{1}$$

Φ is a general indicator of how much traffic to expect in a topology and will serve as metric for total traffic.

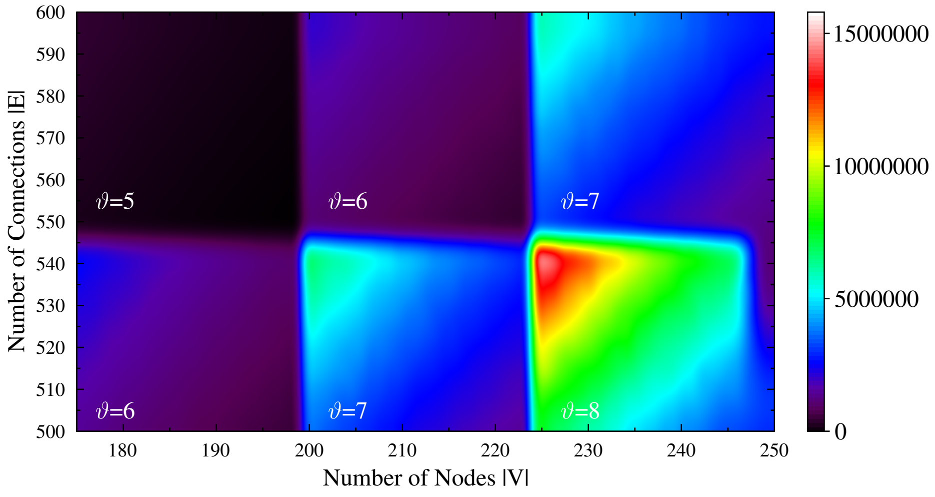
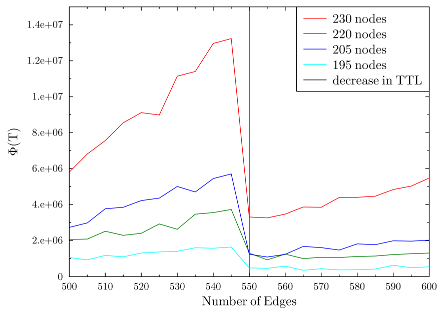
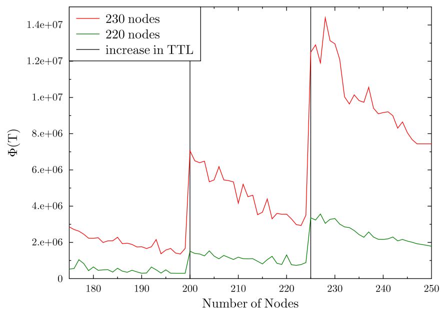


Figure 16: 3D color plot of $\Phi(T)$ depending on $|V|$ and $|E|$, grouped by ϑ



(a) Φ depending on $|E|$, four selections



(b) Φ depending on $|V|$, two selections

Figure 17: Selections of Figure 16

Considering the graph properties of a topology, it should be noted that the effect $|V|$, $|E|$ and ϑ have on Φ is interdependent. For a fixed number of nodes, an increase in the number of connections will generally increase the amount of copies created in a topology and in turn Φ, but only up to the point where the additional connections result in a shorter paths and thus a lower ϑ. Similarly, given a fixed number of connections, an increasing number of nodes will "stretch" the graph, lowering Φ, until the longer path result in a higher ϑ.

Figure 16 shows Φ depending on $|V|$ and $|E|$ and ϑ. As an external constraint, the topologies were chosen in a way that they have the same ϑ for the same $|V|$ and $|E|$, so that topologies with the same ϑ are grouped in a rectangle. Each data point is calculated according to formula (1) and averaged over 50 random topologies, which means that about 75000 topologies were sampled. The range of values is a compromise between giving a overview over a certain area with different ϑ values and the computational time it requires to calculate the data points.
The following observations can be made on Figure 16 and the selections of it in Figure 17:

- Φ generally decreases with an increasing number of nodes $|V|$, given a constant number of connections $|E|$ and a constant ϑ, but not monotonously so
- Φ generally increases with an increasing number of connections $|E|$, given a constant number of nodes $|V|$ and a constant ϑ, but not monotonously so
- The influence of ϑ on Φ is highly significant.
- At least for the two observable cases, similar values for ϑ produce similar ranges of values for Φ

This confirms the considerations above, albeit with one reservation: Figure 16 somewhat clouds the fact that the value of a Φ can vary significantly for different topologies which share the same values for $|V|, |E|$ and ϑ. It is still visible in the selections in Figure 17 that, despite being the average over 50 data points, the resulting correlations are not monotonous. This is because the structure, or connectivity, of the topology is, highly important for Φ. The ratio of nodes to connections is a very simple measure of connectivity, which results in a strong correlation only when sampling many topologies. Therefore, a better measure of connectivity should be established.

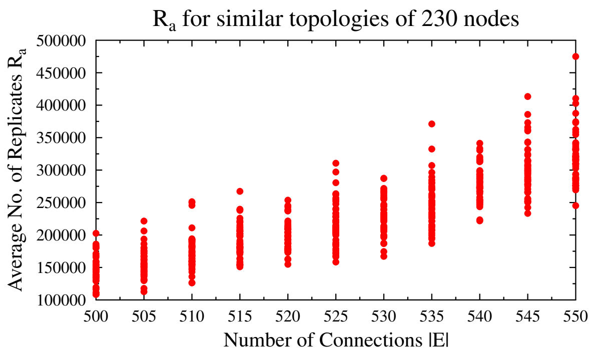
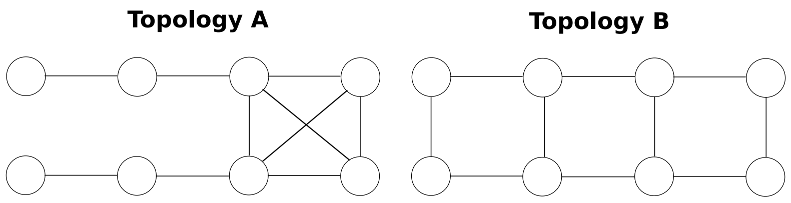
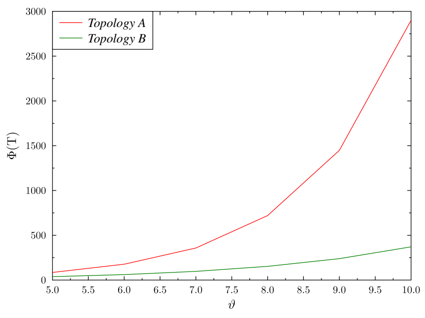


Figure 18: Divergence of Φ among topologies with identical $|V|$, $|E|$ and ϑ.



(a) Topologies A,B with identical $|E|$ and $|V|$, but different connectivity



(b) Φ for A,B plotted against ϑ

Figure 19: Influence of connectivity on Φ

Figure 18 illustrates the problem by showing the range of values Φ can assume for topologies that share the same values for $|V|$, $|E|$ and ϑ. Figure 19 on the other hand show the cause for this: the example topologies A and B have the same amounts of nodes and connections, but differ in structural connectivity. This difference in structure results in significant difference in Φ, especially for a large TTL.

Specifically, topology A contains a complete subgraph, which can be regarded as somewhat of a worst case: given a topology $T = (V, E, \vartheta)$ that contains a complete subgraph $V' \subseteq V$, a packet originating on a start node $v_s \in V'$ within the subgraph will cause at least

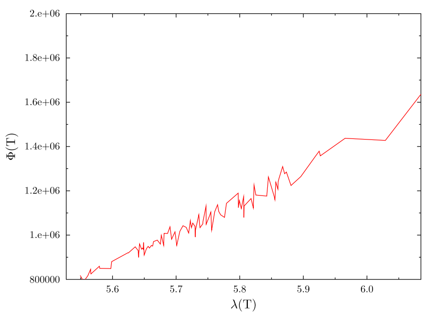
$$\Phi(T, v_s) \geq \sum_{i=0}^{\vartheta} (|V'| - 1) \cdot (|V'| - 2)^{\vartheta}$$

copies to be generated. (The original node sends copies to a minimum of $(|V'| - 1)$ nodes, and each copy causes in turn at least $(|V'| - 2)$ copies to be generated until the time to live reaches zero.)

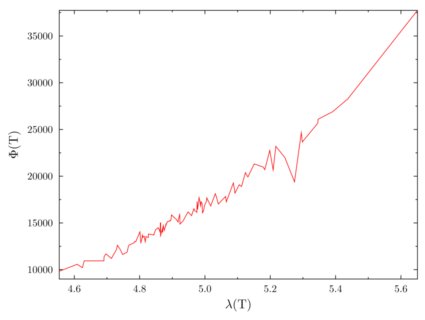
It is therefore necessary to introduce a measure of connectivity that takes into account the presence of densely connected subgraphs. The eigenvector centrality (which is perhaps best known in a variant called Google PageRank [BL06]), and the associated eigenvalue of a graph, are a suitable choice for this. Given a graph $G = (E, V)$, the eigenvector centrality assigns each vertex v a value x_v which denotes its relative importance in the graph and recursively depends on the values for $M(v)$, the set of all vertices adjacent to v: [Wik]

$$x_v = \frac{1}{\lambda} \sum_{v' \in M(v)} x_{v'}$$

where λ is a constant. In other words, the values $x_{v_1}, x_{v_2}, ..., x_n$ form an eigenvector of the graph, and its corresponding eigenvalue is λ. The value of each x_i is a measure of the node's importance in the graph, and the eigenvalue is an overall measure of how well the graph is connected. Since only λ is of interest here, and the Perron-Frobenius theorem proves that the eigenvalue of the centrality eigenvector is always the greatest eigenvalue of a graph [Wik], the following concise definition is sufficient here:



(a) $|V| = 230$, $|E| = 530$



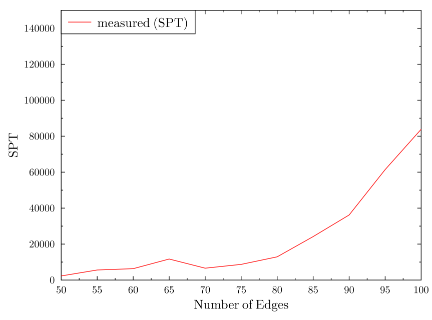
(b) $|V| = 40$, $|E| = 80$

Figure 20: Φ depending on $\lambda(T)$ with fixed $|E|$ and $|V|$

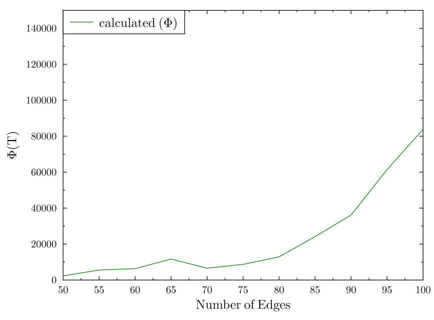
- For a Topology T, let $A = [a_{x,y}]_{n\times n}$ be the complete adjacency matrix. Then $\lambda(T)$, the **greatest eigenvalue** or **connectivity** of T, is the largest $\lambda \in \mathbb{R}$ for which a vector $x \in \mathbb{R}^n$ exists with:

$$Ax = \lambda x$$

All eigenvalues in this thesis are approximated using von Mises-iteration [HK04], one of many numerical methods of available for this problem.
In Figure 20, 100 topologies have been sampled with a fixed $|E|$ and $|V|$ each and the total traffic modifier $\Phi(T)$ has been plotted against the connectivity $\lambda(T)$. The boundaries of the x-axis had to be determined empirically, because the range of the eigenvalue differs greatly depending on $|E|$ and $|V|$. Figure 20(b) is the size of a typical test network in the prototype; Figure 20(a), with 530 nodes and 230 edges, can be directly compared with Figures 16 and especially 18. In both cases, it is clearly visible that there is a clear correlation between the eigenvalue $\lambda(T)$ and Φ, even though it is not strictly monotonous.
Summing up the effects graph properties have on total traffic, it is possible to give the following rule for determining topologies which are favorable for TuneInNet. Given a number of nodes $|V|$ and a number of edges $|E|$, the topology should have a ϑ for which it is possible to minimize $\lambda(T)$. Looking back at Figure 16, the goal can be graphically described as "finding a topology in the lower right corner of a rectangle." This rule was applied for the jitter and scalability measurements below.

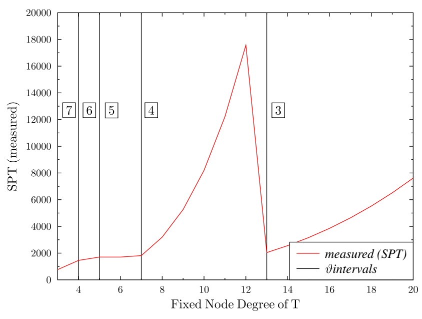


(a) Single Packet Traffic

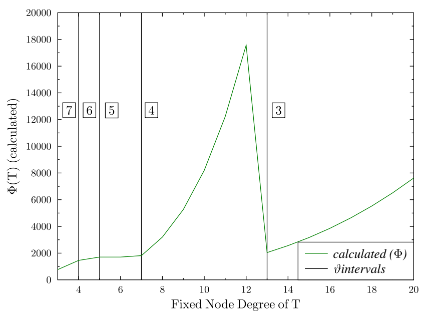


(b) Total Traffic Modifier $\Phi(T)$

Figure 21: Single Packet Test 1, Random Topologies



(a) 40 nodes, no constraints



(b) 40 nodes, fixed node degree

Figure 22: Single Packet Test 2, Topologies with Fixed Node Degree

6.3 Single Packet Test

As already indicated, Φ can also be used to test the implementation. This is because the value of $\Phi(T)$ essentially is the number of packets sent over any link when one packet initially originates at every node, divided by the number of network nodes.

This naturally leads to the following test setup: construct a topology T, send a single packet from every node, and count the total amount of packets sent over any link. That number, divided by $|V|$, should be identical to $\Phi(T)$. If the size of the leaky buckets' queues is chosen sufficiently large, no packet should be lost if not explicitly dropped by the prototype itself.

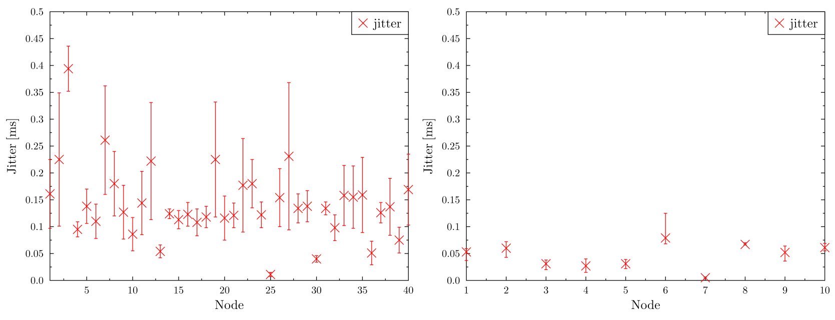
Figure 22(b) shows the results for two setups, with **SPT** being the number of measured packets divided by $|V|$. Both cases were topologies of 40 nodes and had a varying number of edges. In 22(a), the topologies were not subject to any additional constraints. In 22(b), topologies with a fixed node degree were chosen in order to put equal stress on all nodes and increases in ϑ were again fixed to form discrete intervals. The numbers do, in fact, line up perfectly, as can be seen by the respective graphs being basically identical. This means that, as far as this can be inferred from the test of a distributed system, this demonstrates that the prototype works as specified.

6.4 Jitter (Packet Delay Variation)

In the context of networking, jitter generally refers to the difference in delay between pairs of packets in a stream. Jitter is of interest here, because it is a quality of a network's traffic that is important to streaming applications. Knowing the jitter value of a stream allows an application to allocate and manage its buffers. [DC02]
The exact definition of of the metric can vary; Iperf, the tool that was used to measure jitter, implements the definition from [SCFJ02]:

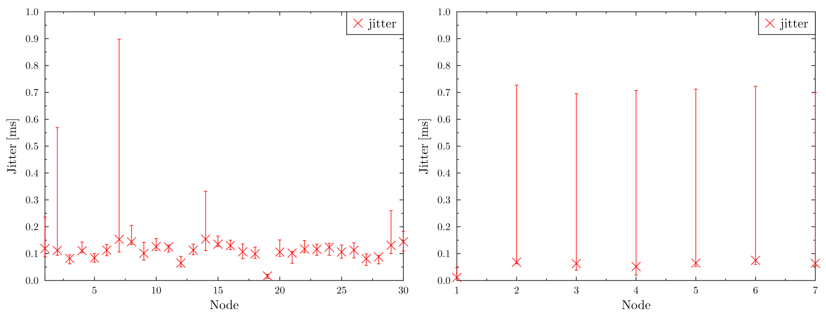
> "The interarrival jitter J is defined to be the mean deviation (smoothed absolute value) of the difference D in packet spacing at the receiver compared to the sender for a pair of packets."

As described in section 5, the physical properties of the system are not completely masked by the prototype, which is why the multicast implementation was added. In this setting, both the TuneInNet flooding and the classical approach of sending the data are subject to roughly equal effects caused by the prototype itself.
Two topologies were created, with 30 and 40 nodes respectively, which is a comfortable range for the limited available hardware. In the case of multicast, about a quarter of the nodes were designated as multicast receivers. A stream of data with a constant bitrate was sent with iperf and received with a different iperf instance on every node in the network.



(a) Topology of 40 nodes, TuneInNet (b) Topology of 40 nodes, Steiner Tree of 10 nodes, Multicast

Figure 23: Jitter Measurement 1, network of 40 nodes



(a) Topology of 30 nodes, TuneInNet (b) Topology of 30 nodes, Multicast, Steiner Tree of 7 nodes

Figure 24: Jitter Measurement 2, network of 30 nodes

Figures 23 and 24 show the result. Comparing the values of 23(a) with 23(b) and 24(a) with 24(b), value ranges for TuneInNet are generally doubled or tripled compared with the multicast implementation. Values for individual differ greatly in TuneInNet and the confidence intervals indicate that these values were also subject to considerable amount variation.

The variation of values among individual nodes for the multicast reference implementation, on the other hand, is very low in comparison. The confidence intervals are also much better. The deviations seen in 24(b) are all the result of a single run, and resulted in a unified different range of values. This makes it likely that this was either caused by background traffic or some other, unexplained effect the hardware had collectively on this single measuring.

The generally larger values for TuneInNet and the higher variation for each value both indicate that its generally high level of traffic results in greater variations in delay. A likely explanation is that in TuneInNet, many more packets compete with each for being flooded by the kernels and copies of the semantically same packet can temporarily dominate the network: If copies of the same packet win the race for a place in a kernel's queue, this has a positive feedback loop as the winning copies create more copies of themselves. While this finding is not favorable for TuneInNet, it is somewhat relativized by the fact that the prototype operates relatively removed from the physical layer, making the significance of measurements depending on precise time measurements preliminary.

6.5 Scalability of the Prototype

The final and most elaborate measurements concerns the scalability of the prototype. It aims to find out how much bandwidth it must provide for applications to function as expected. For this purpose, two exemplary topologies have been chosen. Topology S_1 with a minimum amount of nodes and connections that makes meaningful observations possible, and Topology S_2 at the upper limit of nodes the prototype and available hardware can support. Both have been created with repeated runs of the topology generator until topologies were produced which satisfy the attributes given at the end of the considerations on graph properties. Both have a low ϑ with respect to their number of nodes and connections, and a low λ relative to the possible range of eigenvalues.

6.5.1 Experimental Setup

Each data point is the median over 7 runs. For each of these runs, three streams of data were consecutively send for two minutes from nodes within the network and the packet delivery rate (PDR) was measured. The nodes where the streams originate were fixed for the entire measurements. For each data point, the constant bit rate (CBR) of the streams and the artificial bandwidth (B) of the network were varied within a certain range, which was empirically chosen in a way that the expected PDR lies roughly between 100% and 80%. This is the area of interest, because a PDR below 95% can be seen as an effective breakdown of the network. Each dataset took about 100 hours to produce.

6.5.2 Topology S_1

Topology S_1 has the following graph properties:

Number of Nodes $|V| = 41$

Number of Edges $|E| = 60$

SPL $\vartheta(S_1) = 6$

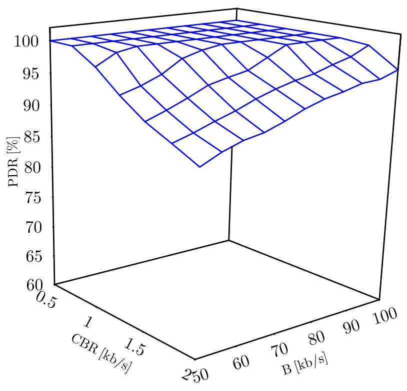
Eigenvalue $\lambda(S_1) = 3.56$

Total Traffic Modifier $\Phi(S_1) = 962.4$

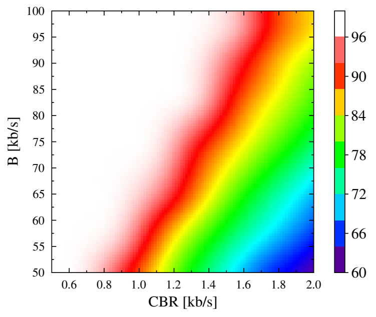
The intervals of measuring were:

$0.5kb/s \leq CBR \leq 2kb/s$ **with steps of** 0.25

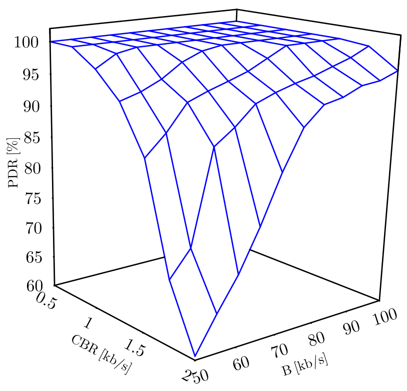
$50kb/s \leq B \leq 100kb/s$ **with steps of** $5kb/s$



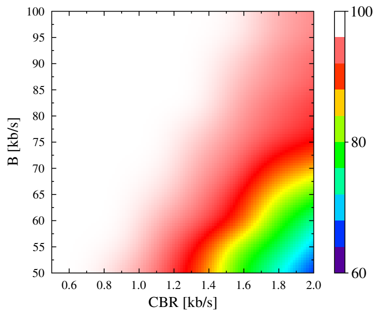
(a) 3D plot,average PDR



(b) 3D color map, average PDR



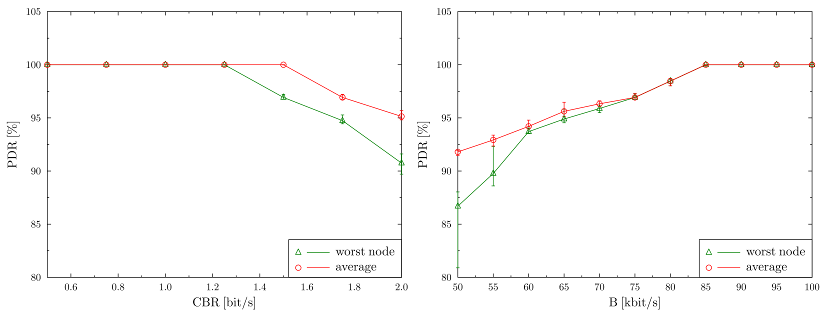
(c) 3D plot,worst case PDR



(d) 3D color map, worst case PDR

Figure 25: PDR for Topology S_1

Figure 25 shows the result for topology S_1. The following observations can be made: firstly, correlations generally exist the way one would expect them to be. For a fixed bandwidth, a stream will have a worse throughput the higher its constant bit rate is, and for a fixed constant bit rate, a stream will have a worse throughput the lower the provided bandwidth is. This is evidenced by the fact that the PDR drops along both axes accordingly. Furthermore, the drop is considerably steeper for the worst case node. This, too, is not surprising, as one would expect central nodes to be forced to drop packets considerably quicker than nodes less connected.



(a) Selection of 25(a) and 25(c) with B = 75 kBit/s and confidence intervals

(b) Selection of 25(a) and 25(c) with CBR = 1.5 kBit/s and confidence intervals

Figure 26: Selections of Figure 25

The error margin, as seen in the "slices" of the graph shown in Figure 26, is rather small, with a single exception in the worst case node. This may be attributable to temporary background network traffic or CPU load on the machines.

The "bird's eye views" (color maps) on the graph seem to suggest a linear correlation between CBR and B. The fact that the drop in PDR appears to follow a similar curve (especially visible in the worst case graph) further supports this, as this indicates that B and CBR have equal weight in this correlation. Taking the measurements from the average case PDR which are placed farthest apart from each other, $(1.25, 50)$ and $(2, 85)$, a linear fit can be applied which basically goes through the green area in Figure 25(b).

This results in a line with a slope of:

$$m = \frac{85 - 50}{2 - 1.25} = 46.67$$

For the worst case, the values are very similar with $(1.25, 55)$ and $(2, 85)$, resulting in a slope of:

$$m = \frac{85 - 50}{2 - 1.25} = 40$$

If these correlation holds, this would mean that for applications to work, the bandwidth provided for this topology would have to be increased 40-fold compared to classical routing. The following measurement of the second topology S_2 will allow to make a conjecture if such a linear correlation generally exists.

6.5.3 Topology S_2

Topology S_1 has the following graph properties:

Number of Nodes $|V| = 100$

Number of Edges $|E| = 175$

SPL $\vartheta(S_2) = 8$

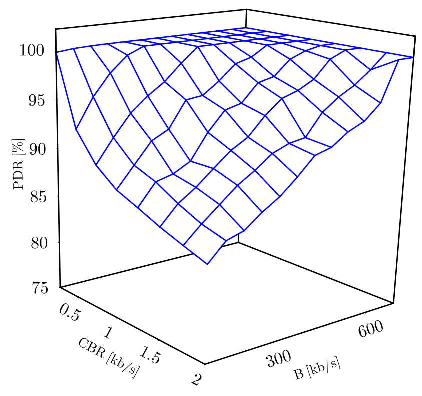
Eigenvalue $\lambda(S_2) = 4.25$

Total Traffic Modifier $\Phi(S_2) = 40223$

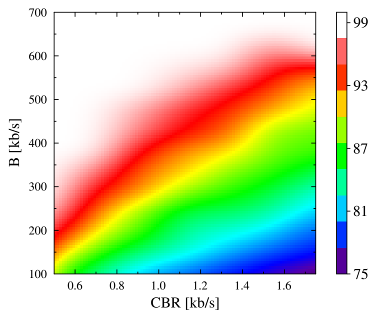
The intervals of measuring were:

$0.5kb/s \leq CBR \leq 2kb/s$ **with steps of** 0.25

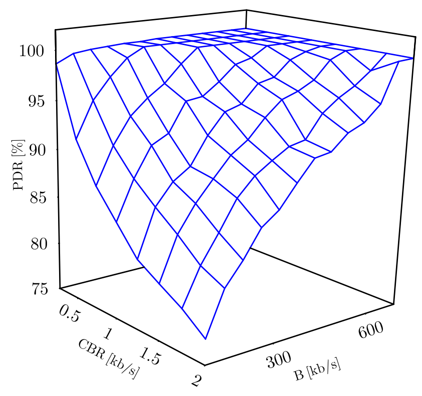
$50kb/s \leq B \leq 100kb/s$ **with steps of** $5kb/s$



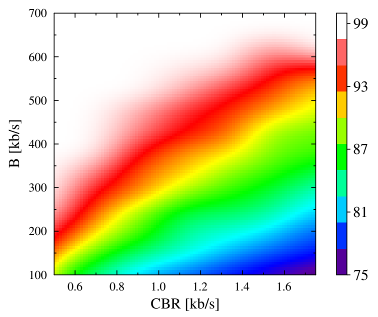
(a) 3D plot,average PDR



(b) 3D color map, average PDR

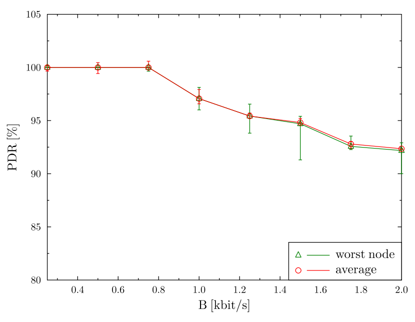


(c) 3D plot,worst case PDR

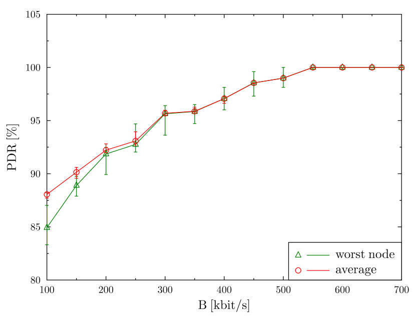


(d) 3D color map, worst case PDR

Figure 27: PDR for Topology S_1



(a) Selection of 27(a) and 27(d) with B = 400 kBit/s



(b) Selection of 27(a) and 27(d) with CBR = 1 kBit/s

Figure 28: Selections of Figure 27

Although the correlations are a bit less pronounced in Figures 27(a) and 27(c), the overall picture is again as one would expect it. The higher variation among PDR values is likely to be caused by the fact that a larger topology is more subject to the effects of random processes and race conditions. When two packets with a different TTL compete for being either further flooded or dropped, dropping the one with a high TTL will lower the total traffic significantly. This reasoning is further supported by the larger confidence intervals in the selections seen in Figure 28.
The graphs for the average and the worst case are also far more similar to each other than was the case for S_1, which means that the average PDR is nearer to the worst case PDR. While the drop along the x-axis, i.e. CBR, appears to be a little sharper in this case, the top-down view still suggest a linear fit again. In this case, the points where the PDR drops below 95% are the same for both best and worst case measurements. With $(0.75, 200)$ and $(2, 550)$, the resulting line has a slope of:

$$\frac{550 - 200}{2 - 0.75} = 280$$

6.5.4 Summary

Since a linear correlation was found in both scalability tests, this suggests that any topology T might have an effective bandwidth $EBR(T)$, that needs to be present for an application that usually needs a bandwidth B to function normally. With a few reservations due to the less clear results from S_2, it can be conjectured that at least for favorable topologies, there might generally exist a constant $K(T)$, so that

$$EBR = K(T) \cdot B$$

If this could be further verified, it would make it easy to calculate the physical bandwidth that must be provided for a TuneInNet network. Ideally, a correlation could be devised between $K(T)$ and $\Phi(T)$. However, if there is a such a correlation between $K(T)$ and $Phi(T)$, it at least appears not to be linear, as Table 3 indicates; a larger set of topologies would have to be sampled with scalability tests in order to find such a relation, which was not possible here, because these tests are highly time-intensive.

$\Phi(T)$	$K(T)$
962.4	40
40223	280

Table 3: Φ and K for S_1 and S_2

On the one hand, it is interesting to note that the magnitude of K is rather small in relation to Φ. The absolute figures for Φ, which have not been discussed so far, may seem overwhelming at first – in topology S_2, a single packet causes about 40223 copies in the network, if the bandwidth is available. However, only the 280-fold bandwidth is required in order to have reasonable PDR rates. The reason is that of those tens of thousands of copies that are potentially created of a packet, only one has to arrive at every node. This means that TuneInNet can operate well below the levels of bandwidth that the rather impressive numbers of Φ seems to suggest at first sight. It also means that a TuneInNet network would practically be always working at maximum load, because any spare bandwidth is always used up by additional copies.
On the other hand, even a 280-fold increase in bandwidth would still be rather uneconomical in a network of 100 nodes, and the topologies S_1 and S_2 were explicitly chosen to have favorable properties. On the scale of the Internet, the constant K may be several orders of magnitude larger, possibly offsetting any router cost saved by using TuneInNet. In other words, on the basis of the scalability tests, the preliminary answer to the question "will the benefits of having simple optical switches outweigh the anticipated cost of the additional bandwidth" that was asked in section 3 is "no" or at least "not without introducing additional constraints into TuneInNet." The final section will outline how the design of TuneInNet might possibly be modified to reduce the total traffic it produces.

7 Summary and Future Work

7.1 Summary

The thesis started out by outlining how current problems of the Internet stem from the fact that its original design did not anticipate what it would turn into, placing an emphasis on the fact that routing and functionality for streaming applications are among the core problems. It then introduced the rationale for clean slate approaches, which seek to rebuild the internet from scratch, and gave an overview over them, establishing that there is a significant interest in them beyond the academic community. Selected alternative paradigms to address-based routing were presented, which generally had in common that they tried to redesign or relocate the network's complexity. Contrary to this consensus, an argument was presented that routing might be the consequence of a flawed fundamental design choice. Based on this argument, TuneInNet's alternative, flooding and filtering, was detailed and a prototype architecture was developed, which consists of three components: a kernel, which floods traffic, a data packet format for the flooding, and an API which provides basic methods for sending and receiving data to applications.

The architecture was implemented on GNU/Linux, where the kernel takes the form of a server daemon and the API the form of a shared library, so that the prototype could be evaluated with existing hard- and software. An elaborate evaluation setup was added to the prototype to handle large networks and circumvent the physical limits of the available hardware.

The evaluation showed that TuneInNet, at least for the scale that was tested, works in principle and is also usable for multimedia applications, although the jitter measurements cast some doubt on its ability to provide traffic with a stable delay. The effect that the graph properties of a network topology has on the traffic in TuneInNet was regarded analytically in some detail, and a general rule was found to determine topologies which are favorable for TuneInNet's approach.

Two such favorable topologies, S_1 and S_2, were chosen with this rule and systematically tested to find out under which load they fail for a given bandwidth.

The result of these scalability test indicates that TuneInNet may need more bandwidth by a constant factor K than classical networks, if they want to provide the same effective bandwidth – although this factor may turn out to be very large, if no additional measures lower the total traffic in TuneInNet. The core result of this thesis can therefore be summed up as follows: while TuneInNet does work in principle, it is unlikely to be applicable on a large scale in its most radical form, as it would likely increase the need for bandwidth in an amount that would not be offset by the advantage of not needing routers. In the following, final remarks, a few ideas are presented on how the traffic in TuneInNet could be contained.

7.2 Future Work and Final Remarks

This thesis deliberately excluded the filters proposed in its architecture and evaluated TuneInNet in its most radical form. This was done, because filters are the point where complexity may re-enter into the network design. The results of the scalability tests suggest that the task to develop filters that do not introduce significant amount intelligence into the network may be very difficult. Ideally, filters could generally be restricted to simple concepts like grouping traffic as belonging to geographic regions, but this may not be enough. The most pressing open question is therefore how dynamically configurable filters of optical switches could be without compromising the ideal of a very simple network device.

Two possible additional mechanism that could lower the total traffic significantly are the dynamic negotiation of TTL values and more specific measures for designing suitable topologies. Dynamic TTL negotiation would mean that an application that streams data may periodically check how many hops its farthest listener is away and then set the TTL accordingly. Similarly, point-to-point applications may establish the lowest TTL necessary for stable communication upon connecting. This would, in accordance with TuneInNet's basic approach, be a mechanisms located in the endpoints. More sophisticated metehods for designing topologies might include intentionally adding specific redundant paths between nodes with a big hop distance in order to reduce the necessary TTL as well as splitting the network into subnets.

In any case, further research will require experiments on a larger scale. The prototype implementation developed and presented here is a proof of concept for comparatively small networks, but reasonable statements about a larger scale are not possible without experimentation on that scale.
Finally, it may be reasonable to explore different scenarios in which TuneInNet could be useful, such as broadcasting within an Autonomous System, or on limited parts of a topology. Given the increasing amount of streaming applications on the Internet, a broadcasting infrastructure similar to the MBone might be proposed and further research into broadcasting on the Internet backbone is certainly warranted.
In conclusion, the result of this thesis may be tentatively negative, but this is no final verdict on the attempt to reduce the network's complexity. The empirical results of this thesis give an estimate about how the pure version of TuneInNet behaves, and future, less radical variants can now built upon that. Devising a way to reduce traffic without re-introducing too much intelligence into the network is a demanding task, but may still yield part of the design of the Future Internet. As Leslie Daigle recently remarks in the IETF Journal – "If you can predict the future of the Internet, it's no longer the Internet." [Dai09]

Bibliography

[BL06] Kurt Bryan and Tanya Leise. The \$25,000,000,000 Eigenvector: The Linear Algebra behind Google. *SIAM Review*, 48:569 – 581, 2006.

[Bla02] Uyless Black. *MPLS and Label Switching Networks*. Prentice Hall PTR, 2002.

[Blu00] Daniel J. Blumenthal. All-Optical Label Swapping for the Future Internet. *Optics & Photonics News*, 13, 2000.

[Car96] B. Carpenter. RFC 1958: Architectural Principles of the Internet, 1996. `http://www.ietf.org/rfc/rfc1958.txt`.

[Car04] Gianni Di Caro. *Theory and Practice of Ant Colony Optimization for Routing in Dynamic Telecommunications Networks*. PhD thesis, Université Libre de Bruxelles, 2004.

[CDG$^+$09] Vint Cer, Bruce Davie, Albert Greenberg, Susan Landau, and David Sincoskie. FIND observer panel report, 2009. `http://www.nets-find.net/FIND_report_final.pdf`.

[Cha07] Vincent Chan. NeTS-FIND - Future Optical Network Architectures (FIND Project description), 2006/2007. `http://www.nets-find.net/Funded/file_download.php?file=pdf_files/FutureDesc.pdf`.

[Cro03] Jon Crowcroft. Turing Switches - Turing machines for all optical Internet routing. Technical Report 556, University of Cambridge, 2003. `www.cl.cam.ac.uk/techreports/UCAM-CL-TR-556.pdf`.

[CSSS07] Georg Carle, Jochen Schiller, Andreas Schrader, and Bernhard Szz. Method for forwarding data packets in a network and communication network having flooding transport properties (International Application No.: PCT/EP2007/005244), 2007.

[Dai09] Leslie Daigle. Developing internet technology research and standards. *IETF Journal*, September 2009.

[DC02] C. Demichelis and P. Chimento. RFC 3393: IP Packet Delay Variation Metric for IP Performance Metrics (IPPM), 2002. `http://www.ietf.org/rfc/rfc3393.txt`.

[DH98] S. Deering and R. Hinden. RFC 2460: Internet Protocol, Version 6 (IPv6) specification, December 1998. `http://www.ietf.org/rfc/rfc2460.txt`.

[Dor92] Marco Dorigo. *Optimization, Learning and Natural Algorithms.* PhD thesis, PhD thesis, Politecnico di Milano, Italie, 1992.

[FBR+04] Nick Feamster, Hari Balakrishnan, Jennifer Rexford, Aman Shaikh, and Jacobus van der Merwe. The Case for Seperating Routing from Routers. *Applications, Technologies, Architectures, and Protocols for Computer Communication archive Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 5–12, October 2004.

[Fel07] Anja Feldmann. Internet Clean-Slate Design: What and Why? *ACM SIGCOMM Computer Communication Review*, 37:59–64, 2007.

[Fis07] Darleen Fisher. US National Scienence Foundation and the Future Internet Design. *ACM SIGCOMM Computer Communication Review*, 37:85 – 87, 2007.

[GGZ07] Roch Guerin, Lixin Gao, and Zhi-Li Zhang. A Framework for Manageability in Future Routing Systems (FIND project description), 2006/2007. `http://www.nets-find.net/Funded/file_download.php?file=pdf_files/FrameworkDesc.pdf`.

[GHM+05] Albert Greenberg, Gisli Hjalmtysson, David A. Maltz, Andy Myers, Jennifer Rexford, Geoffry Xie, Hong Yan, Jibin Zhan, and Hui Zhang. A Clean Slate 4D Approach to Network Control Managment. *ACM SIGCOMM Computer Communication Review*, 35, October 2005.

[HK04] Hans R. Schwarz and Norbert Köckler. *Numerische Mathematik.* 2004.

[HZZL09] Ning Hu, Peng Zou, PeiDong Zhu, and Xin Liu. Cooperative Management Framework for Inter-domain Routing System. *Autonomic and Trusted Computing*, pages 567 – 576, 2009.

[IPE] Iperf. `http://sourceforge.net/projects/iperf/`.

[Jar02] Jared Winick and and Cheng Jin and Qian Chen and Sugih Jamin. Inet Topology Generator, 2002. `http://topology.eecs.umich.edu/inet/`.

[KCF07] Dmitri Krioukov, KC Claffy, and Kevin Fall. Greedy Routing on Hidden Metric Spaces as a Foundation of Scalable Routing Architectures without Topology Updates (FIND project description), 2006/2007. `http://www.nets-find.net/Funded/file_download.php?file=pdf_files/Greedy.pdf`.

[KMB81] L. Kou, G. Markowsky, and L. Berman. A fast algorithm for steiner trees. *Acta Informatica*, 15, 1981.

[Mac94] Michael Macedonia. MBone Provides Audio and Video Across the Internet. *IEEE Computer*, 4:30 – 36, 1994.

[MD07] Pedro Veiga Monica Domingues, Carlos Friacas. Is global ipv6 deployment on track? *Internet Research*, 17:505–518, 2007.

[Meh88] Kurt Mehlhorn. A faster approximation algorithm for the steiner problem in graphs. *Information Processing Letters*, 27, 88.

[MZF06] D. Meyer (Editor), L. Zhang (Editor), and K. Fall (Editor). RFC 4984: Report from the IAB Workshop on Routing and Addressing, 2006. `http://www.ietf.org/rfc/rfc4984.txt`.

[NBBB98] K. Nichols, S. Blake, F. Baker, and D. Black. RFC 2474: Definition of the Differentiated Services Field (DS Field) in the IPv4 and IPv6 headers, December 1998. `http://www.ietf.org/rfc/rfc2474.txt`.

[Per02] C. Perkins (Editor). RFC 3344: IP Mobility Support for IPv4, 2002. `http://www.ietf.org/rfc/rfc3344.txt`.

[SA09] Fabrizio Sestini and Susanna Avéssta. FIRE White Paper, 2009. http://www.ict-fireworks.eu/fileadmin/documents/FIRE_White_Paper_2009_v1.02.pdf.

[SCE$^+$05] Lakshminarayanan Subramanian, Matthew Caesar, Cheng Tien Ee, Mark Handley, Morley Mao, Scott Shenker, and Ion Stoica. HLP: A Next Generation Inter-domain Routing Protocol. *ACM SIGCOMM Computer Communication Review*, 4:13 – 24, October 2005.

[SCFJ02] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RFC 3993: RTP: A Transport Protocol for Real-Time Applications, 2002. http://www.ietf.org/rfc/rfc3393.txt.

[VLC] VLC Media Player. http://www.videolan.org/vlc/.

[Wik] Wikipedia. (URL) Wikipedia Article on Centrality. http://en.wikipedia.org/wiki/Centrality.

[Win87] P. Winter. Steiner problem in networks: a survey. *Networks*, 17, 1987.

[YMBB05] M. Yannuzzi, X. Masip-Bruin, and O. Bonaventure. Open issues in interdomain routing: a survey. *IEEE Network*, 19:49–46, Nov-Dec. 2005.

List of Figures

List of Tables

A Sample XML Configuration File

Listing 1: "Sample TuneInNet configuration file"

```
<?xml version="1.0"?>
<TuneInNet>
<!-- topology generator parameters: -->
<!-- random seed: 185099 size: 10 mindegree: 2 maxdegree: 9999 -->
<!-- bucket size: 5000 bits per second: 80000 TTL: 5-->
<!-- Eigenvalue:3.582593: -->
<!-- Reproduce with: -r 185099 -s 10 -E 15 -d 2
<!--    -e 9999 -B 80000 -S 5000 -T 5 -M 0 -->    -->
<General>
<Mode> TUNEIN </Mode>
    <Protocol>TCP</Protocol>
    <Logdir>/TI/log</Logdir>
    <Loglevel>CRITICAL</Loglevel>
    <TTL>5</TTL>
<LeakyBucket>
        <LBBS>80000</bitss>
        <LBSize>5000</size>
</LeakyBucket>
</General>

<Nodelist>
<Node ti_name="host1.0" ip="10.0.0.1" port="5566"
      ti_label="0" multicast="y" />
<Node ti_name="host2.0" ip="10.0.0.2" port="5566"
      ti_label="1000" multicast="y" />
<Node ti_name="host3.0" ip="10.0.0.3" port="5566"
      ti_label="2000" multicast="y" />
<Node ti_name="host1.1" ip="10.0.0.1" port="5567"
      ti_label="1" multicast="y" />
<Node ti_name="host2.1" ip="10.0.0.2" port="5567"
      ti_label="1001" multicast="y" />
<Node ti_name="host3.1" ip="10.0.0.3" port="5567"
      ti_label="2001" multicast="n" />
<Node ti_name="host1.2" ip="10.0.0.1" port="5568"
      ti_label="2" multicast="n" />
<Node ti_name="host2.2" ip="10.0.0.2" port="5568"
      ti_label="1002" multicast="n" />
<Node ti_name="host3.2" ip="10.0.0.3" port="5568"
```

```
       ti_label="2002" multicast="n" />
<Node ti_name="host1.3" ip="10.0.0.1" port="5569"
       ti_label="3" multicast="n" />
</Nodelist>

<Topology>
<Connection node1="host1.0"
     node2="host3.1" bandwidth="5" />
<Connection node1="host1.0"
     node2="host1.3" bandwidth="5" />
   <Connection node1="host2.0"
       node2="host1.1" bandwidth="5" />
  <Connection node1="host2.0"
       node2="host2.1" bandwidth="5" />
  <Connection node1="host2.0"
       node2="host3.1" bandwidth="5" />
  <Connection node1="host2.0"
       node2="host2.2" bandwidth="5" />
  <Connection node1="host3.0"
       node2="host1.3" bandwidth="5" />
  <Connection node1="host1.1"
       node2="host2.1" bandwidth="5" />
  <Connection node1="host1.1"
       node2="host3.1" bandwidth="5" />
  <Connection node1="host1.1"
       node2="host1.2" bandwidth="5" />
  <Connection node1="host3.1"
       node2="host1.2" bandwidth="5" />
  <Connection node1="host3.1"
       node2="host2.2" bandwidth="5" />
  <Connection node1="host3.1"
       node2="host3.2" bandwidth="5" />
  <Connection node1="host1.2"
       node2="host3.2" bandwidth="5" />
  <Connection node1="host3.2"
       node2="host1.3" bandwidth="5" />
</Topology>
</TuneInNet>
```

Internet Society Copyright Notice

Hiermit versichere ich, dass ich die vorliegende Arbeit selbständig angefertigt und ohne fremde Hilfe verfasst habe, keine außer den von mir angegebenen Hilfsmitteln und Quellen dazu verwendet habe und die den benutzten Werken inhaltlich oder wörtlich entnommen Stellen als solche kenntlich gemacht habe.

Berlin, den

Sebastian Dill