Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik
und Kommunikationsnetze
Computer Engineering and Networks Laboratory

Thomas Other

Opportunistic Networks with ANA

Master Thesis
MA-2008-23

Department:
TIK – Computer Engineering and Networks Laboratory, ETH Zurich

Supervisor:
Prof. Dr. Bernhard Plattner, ETH Zurich

Advisors:
Dr. Franck Legendre, ETH Zurich
Theus Hossmann, ETH Zurich

Zurich, March 2009

Contents

List of Figures

List of Tables

Abstract

Title: Opportunistic Networks with ANA

Keywords: Podcasting, Opportunistic Networking, Delay Tolerant Networking, Autonomic Network Architecture, Embedded Device

With the advent of mobile computing, the Internet has not only become ubiquitous, but also completely new ways of networking became possible. Modern handheld devices implement different networking technologies (such as WiFi and Bluetooth), which are independent of network infrastructure, and can connect as mobile ad-hoc networks to other mobile devices. There is no permanent connectivity in such a mobile scenario, due to the limited range of radio communication and the individual mobility of each participant, therefore it is also termed an opportunistic network.

To distribute information in mobile ad-hoc networks, todays services still rely on protocols inherited from the early days of the Internet, but these protocols might not be optimally suited for recent scenarios. For example the PodNet application, designed to exchange podcast episodes opportunistically in mobile ad-hoc networks, could profit from a more efficient networking approach, where protocols are optimized to suit the mobile scenario. The ANA framework provides such a flexible network stack, that seeks optimal solutions for different networking scenarios. However, up to date, the ANA framework is mainly used for wired and connected networks.

This thesis enables the ANA framework for disruptive tolerant network services, providing a basis to opportunistic networking applications. As an example of such an application we present the implementation of PodNet in the ANA framework, running on handheld devices.

The resulting framework for delay tolerant networking (DTN) applications in ANA is ready to serve as an experimental platform to gain further insight in information dissemination in opportunistic networks. The fact that the information spreading is influenced by individual social behavior, as users carry their mobile computers, will have a big influence on future work in this field.

Acknowledgments

I would like to thank Prof. Dr. Bernhard Plattner for allowing me to do this work at the Communinucation Systems Laboratory at the ETH Zurich. Furthermore I would like to thank Dr. Franck Legendre and Theus Hossmann, my advisors, for their support and guidance and great assistance during the work.

Chapter 1

Introduction

During the last dozen of years the Internet enjoyed an ever growing number of users from all over the world. It has, indeed, come a long way from a tool being mainly of commercial and scientific interest, to a network with a permanently increasing pool of ideas and non-commercial services available to its users. Prominent examples are file sharing platforms, like e.g. BitTorrent [1], or online friendship communities as Facebook [2].

With the advent of mobile computing in recent years, the Internet is on the verge to yet another drastic change. In most cellular networks access to the Internet has been readily available for some time now, but the capabilities of mobile devices were too limited to provide a convincing alternative to the well established personal computers. Only very recently mobile computing devices, like for example Apple's IPhone, successfully started to appear on the market, and with such devices the Internet now truly becomes ubiquitous.
Apart from contributing to the Internet's success story, the new generation of mobile computers do also open the door to a different kind of networking itself. Modern devices have in common that they unite different kinds of network technologies, such as the cellular network, WiFi or bluetooth, which can be used to establish individual contacts outside the realm of the global, connected Internet. A network resulting from such individual contacts is often termed *mobile ad hoc network* (MANET), and as its participants move independently, its topology is subject to permanent change. Initially MANETs were assumed to be connected (i.e., there is an end-to-end path between all nodes), but more recent studies in this field revealed, that the assumption of permanent availability of end-to-end paths is not realistic. This led to the paradigm of disruption-tolerant and opportunistic networks, where the mobility of its users is used to disseminate data. To support opportunistic scenarios, new models for routing and data forwarding need to be taken into account.

[1]http://www.bittorrent.com
[2]http://www.facebook.com

The possible range for mobile applications that rely on such networking is large. For example in public transportations, in order to connect all vehicles on route, or more generally speaking in regions where no other network coverage is available. In contrast to these rather sparsely populated scenarios, a city full of people, each carrying a mobile computer, implies a whole new world. It is, indeed, this new world that is about to promote the Internet in the near future, and lead into a truly mobile era.
The major change to todays networks and its users will be, that it will not only serve to form purely virtual communities, but also allow to integrate those virtual communities into our physical presence. Prominent examples would be friendship communities, where friends from Facebook would meet on the street, or partnership platforms, where matches for possible candidates could be immediately inspected. The aka aki [1] platform is a perfect example of such a mobile application, with much resemblance to Facebook, but with the difference that online contacts get promoted to the street, with handheld devices informing their users if friends are within reach. And as the popularity and variety of Internet services suggest, with its number of applications ever growing, such a new kind of mobile netwok would have substantial influence on social patterns in modern society.

As has been explained, the potential for new mobile services is huge, and therefore a framework that supports such applications, but without the burden of the Internet protocols legacy, is the goal of this thesis. The capabilities of this framework are then demonstrated by implementing the opportunistic podcasting application PodNet [2].

1.1 Opportunistic Networking

The mobile networks known as MANETs are only part of the current research in the field of *delay tolerant networking* (DTN). Rather DTN addresses all those networks, that have in common, that they may lack continuous network connectivity, in contrast to the assumptions valid in the Internet.
Although a mobile network, that is driven by people, might on a first impression look purely random, DTN identifies common patterns, and helps towards understanding underlying procedures.

Opportunistic networking refers to a DTN scheme, where it's impossible to predict future encounters, and where any participant may vanish instantaneously. This does exactly describe the kind of mobile network found in the introduction, and is the basic assumption of PodNet. To support such an environment an application must be able to autonomically identify new contacts, check them for potentially available information, and finally retrieve that information. The way this information is distributed in these networks is of primary interest. The simplest version of this scheme is the one-hop opportunistic networking. It describes a nonparticipatory dissemination scheme, where information is exchanged on a direct basis with available contacts, and no assumptions are made about the reachability of other, currently absent, contacts.

1.2 Thesis Description

Amongst Internet services, podcasting has become a very popular and successful service in a short time, as it offers information in a way similar to a subscription to a periodical newspaper. This success illustrates the interest for participatory broadcasting, and motivates the pursuit of further research in this field.
Podcasting, in its earlier form however, was only available with fixed infrastructure support to retrieve publicized episodes. The PodNet project [2] released this limitation by offering a podcasting system based on opportunistic wireless networking, that extends podcasting to ad hoc domains. Consequentally the platform was to be mobile computers, to take advantage of their mobility, and observe the results to support research in the field of mobile networking.

The original PodNet version [2], however, is still tightly bound to the network protocol stack inherited from the Internet version of the service. But these mechanisms might not be optimally suited for a mobile scenario, as mechanisms that connected the global Internet are designed for many more participants than in a DTN scenario, resulting in a big overhead and suboptimal protocol usage, therefore a more flexible use of different network technologies and protocols is required. The *autonomic network architecture* (ANA) project [9] offers such a flexible network stack, and allows the use of different kinds of networking technologies to be operated in parallel. ANA achieves this flexibility by using functional blocks, rather than a fixed protocol stack, that are dynamically arranged to obtain optimal results for different kinds of networking scenarios. Yet, PodNet is only one kind of application that can benefit from opportunistic communications. Therefore this thesis is about the definition and implementation of new communication facilities that enable the development of new opportunistic-based applications, and shall rely on a implementation of PodNet in the ANA framework.

To give a glimpse of the contributions this thesis provides, the following list shall summarize the items implemented in the ANA framework:

- Generic DTN Functions
  - Neighbor Discovery: Explores the network neighborhood, by maintaining a list of nodes within transmission-range (i.e. opportunistic contacts), and provides the set of discovered contacts to the framework.
  - Synchronization Service: Ensures that all peers are aware of content synchronicity, and points opportunistic network applications to potential updates.

- Podcasting Function for Embedded Devices
  - PodNet on ANA: The core podcasting service is mainly composed of elements originally devised in [2], but adapted to the ANA framework. This constitutes the exemplary application for the DTN framework.
  - Transmission Control Protocol: Offers reliable network transmission, and since ANA was not ready to provide such a service, it needed to be implemented.
  - Cross-Compilation: The OpenEmbedded toolchain [3] is used to facilitate platform independence for the ANA framework, and was used to demonstrate the PodNet on ANA application with handheld devices. As a result the Nokia N810, and the Sharp SL-C860 devices were both successfully running the application.

The implementation of a podcasting service, and the necessary communication facilities, in a flexible network architecture like ANA provides several advantages. The design of ANA is better suited for a one-hop opportunistic network application like PodNet, than the Internet protocols, because it has the ability to dynamically rearrange network functionality, and therefore adapt to different network types on the fly. By different network types, not only technological differences are considered, also the ANA framework seeks to adapt its protocol structure, to obtain optimal performance for a given scenario.
Another motivation is the ANA framework itself, its main purpose being the research resulting from a flexible network protocol stack. As the introduction pointed out, the Internet in its present form, will not be able to support this new kind of mobile networks, therefore the ANA framework provides a valuable platform to investigate and to compare results against model predictions. And furthermore, it may help answering a question closely related to such mobile networks, of how exactly such a network of mobile devices, carried by individuals and therefore also a social network, might look like.
To be able to study the behavior of the ANA framework, it needs to be populated with networking applications, that make use of its advantages, and provide an evaluation method for the underlying theoretical concepts. Thefore the PodNet application represents an attractive candidate, and the dissection of the monolithic PodNet application, into smaller blocks, will provide a basis for further research in this field.

1.3 Related Work

There exist several examples of scientific and commercial projects that are related to the subject of this thesis. The "Click Modular Router" by Kohler et al. [4], for example, is a software architecture for building flexible and configurable routers, and bears many analogies to the ANA project. For instance, functions are broken down into smaller building blocks, which can then easily be used to flexibly

compose a new set of network functionalities. Furthermore both projects seek the ability to adapt to changes in the network, in order to maintain a resilient networking service. In contrast to ANA, however, the Click project is limited to routing and packet forwarding mechanisms, whereas ANA encompasses many more aspects to networking.
Another interesting paper by Kawadia et al. [5] deals with "System Services for Ad-Hoc Routing: Architecture, Implementation and Experiences", this work explores several systemic issues regarding the design and implementation of routing protocols for ad-hoc wireless networks. It's focus lies on a general modification of the current IP routing architecture, and a specific implementation of this architecture in Linux, where the key concept is to store packets until a route has been discovered, and not to drop it just because no forward path is available, as done by most IP implementations. More DTN routing literature will be discussed in chapter 2.
The *Delay-Tolerant Network Research Group* (DTNRG) [6], which is a part of the *Internet Research Task Force* (IRFT), has been researching DTN since 2002, and besides numerous publications also developed two protocols. The bundle protocol is a general overlay network protocol, and is the focus of implementation efforts in DTNRG. The Licklider transmission protocol (LTP - sometimes called the longhaul transmission protocol), on the other hand, is a point-to-point protocol supporting very high delay links, such as those used in deep space communications.
In comparison to the previously mentioned projects, the 7DS system [7] bears a very strong resemblance to the subject of this thesis. Indeed, the software provides information exchange applications, that support wireless ad-hoc networks, where persistent end-to-end links may be absent. In comparison to this thesis, however, the 7DS system focuses on the application layer in the network, or said in another way, 7DS uses a top-down method, whereas ANA is trying to build a new network from the bottom-up.
Another similar project to this thesis, the Haggle autonomic network architecture [8], which like ANA breaks with the classical network stack approach, and provides a set of novel mechanism to support intermittent network connectivity.

As an example of a commercial application, the aka aki service [1] shows the available potential to create compelling mobile networking applications. Aka aki offers software each user can download to his mobile phone, and that links to both, the aka aki Internet service, and to other people that are within reach of each other. Aka aki basically provides a friendship platform similar to Facebook, but this service also allows to compare encounters made by the mobile phone with the users list of friends in the Internet. If a match is found, both individuals are notified, that one of their friends is in the vicinity.

1.4 Thesis Outline

The thesis sets out by describing those theoretical aspects, that form the basis of the implementation of PodNet on ANA. After a brief introduction to DTN, chapter 2 summarizes some results from the field of DTN routing and content dissemination. Then it introduces the main concepts inherent in the ANA framework and explains PodNet's strategy of opportunistic content dissemination. Finally this chapter concludes by a description of the transmission control protocol (TCP), that will form the basis of reliable transmission in ANA.
To set the stage for the implementation, chapter 3 analyzes the problems that the implementation must face, and proposes possible solutions. Besides the integration aspects related to ANA, this chapter also gives a brief overview of mobile computing devices, and the question of how to run an application on different kinds of hardware.
A detailed description of all contributions is given in chapter 4. First the functional blocks, that were developed during this thesis, are described in detail. Then the messages passed between these functional blocks are explained, and a description of the relation between these blocks is given. In order to validate the work, the chapter also includes the implementations results and measurements illustrating the performance.
Finally chapter 5 formulates a conclusion, and by looking at possible extensions to the framework, points to possibilities for future projects.

To take some load of the implementation description in chapter 4, the exhaustive listing of function interfaces and internal processes has been moved to appendix A, additionally the instructive details for compiling and installing the software for mobile devices is given in appendix B.

Chapter 2

Fundamentals

The theoretical aspects of delay-tolerant networking are introduced in chapter 2.1, including the analysis of different scenarios. Although the mathematical formulas presented in this chapter are not dominant in the implementation of an opportunistic networking application, these concepts help to establish a basic terminology, and provide rough estimates for different scenarios.
The second section 2.2 covers the basic concepts applied in the *autonomic network architecture* (ANA) [9], and provides a brief overview of the frameworks functional capabilities. Chapter 2.3 describes the application for the 'Wireless Ad Hoc Podcasting Network', abbreviated PodNet [2], and takes a look at its content-dissemination strategy. Finally chapter 2.4 describes the widely used *transmission control protocol* (TCP) [10] in some detail, as an example of a reliable transmission protocol, and as the basis for its implementation in the ANA framework.

2.1 Delay Tolerant Networking

Delay-tolerant networking (DTN) is an approach to computer network architecture that seeks to address the technical issues in heterogeneous networks that may lack continuous network connectivity. Examples of such networks are those operating in mobile or extreme terrestrial environments, or for example over long interplanetary distances in space, where the propagation delay is much larger than transmission times. In general, DTN connections can be categorized into *scheduled* or *predictable* contacts and *intermittent* or *opportunistic* contacts.

Depending on the mobility of the networks members, the lack of continuity in a network connection does not only show in the absence of persistent end-to-end paths, but often also in a segregation of large parts of the network into smaller disconnected subnetworks, as shown in figure 2.1. A delay-tolerant network must thus be able to carry data even if its destination is currently not reachable, furthermore the network must decide which path will be the quickest, in terms of delay, or the most reliable, in terms if delivery ratio, for the data to reach its destination.

Additionally to these challenges many nodes in the network may also have limited resources, be it bandwidth, storage space or energy constraints.

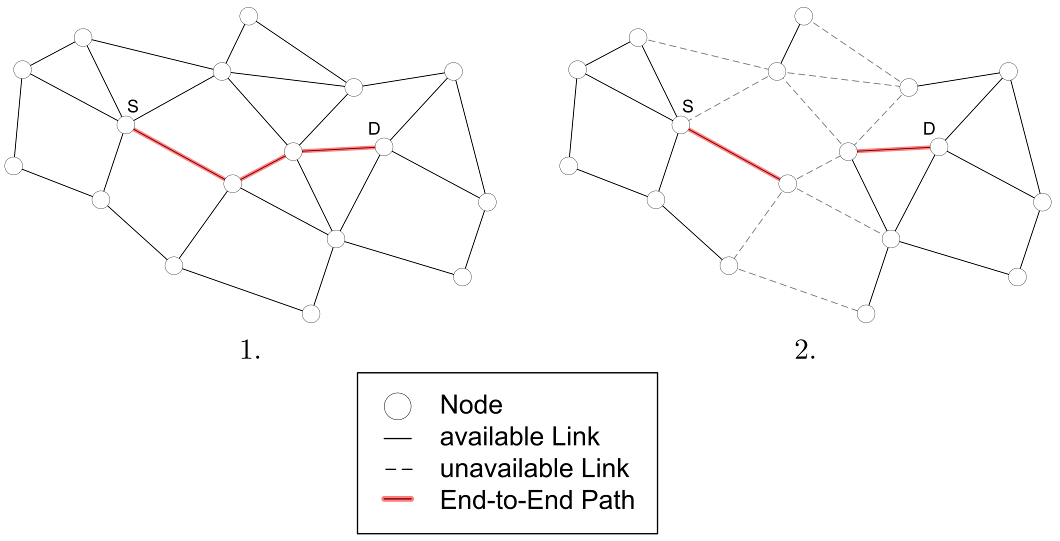


Figure 2.1: Network Graphs: 1. Continuous end-to-end paths, 2. Disrupted end-to-end paths

2.1.1 Routing - A Graph Theory

While there are many characteristics to routing protocols, a convenient way to create a taxonomy especially for DTN routing is based on whether or not the protocol creates replicas of messages. Routing protocols that never replicate a message are considered *forwarding-based*, whereas protocols that do replicate messages are considered *replication-based*. This simple, yet popular, taxonomy was recently used by Balasubramanian et al. to classify a large number of DTN routing protocols [11]. Additionally the use of graph theory reveals basic network properties and offers a consistent way to establish estimates on network parameters for different scenarios.

A DTN graph G is disconnected and/or time-varying, it contains a set of nodes V that represent network entities and a set of edges E connecting these entities, as shown in figure 2.2. Alternatively, the edges may also be characterized as connections C,

$$G(t) = \{V, E(t)\}\,,\ E(t) = set\ of\ edges\ e_i\ at\ time\ t$$

$$G(t) = \{V, C(t)\}\,,\ C(t) = set\ of\ connections\ c_i\ at\ time\ t \tag{2.1}$$

$$c_i = \{v_i, v_j, t_{start}, t_{finish}, bandwidth, propagation\ delay, etc.\}\,. \tag{2.2}$$

A connection c_i is an extension to an edge e_i as it does not only describe the edges end points and the edge weight, but adds network related information to it, as shown in equation 2.2. Therefore a connection c_i may contain detailed information on scheduled or probabilistic contacts, as with e.g. a satellite link or a bus on its route.
Graph theory thus facilitates to solve routing problems, and provides an analytic way to find optimal routes for data packets in deterministic scenarios. There exist several metrics than can be applied to the graph, as for instance delay optimization or troughput optimization.

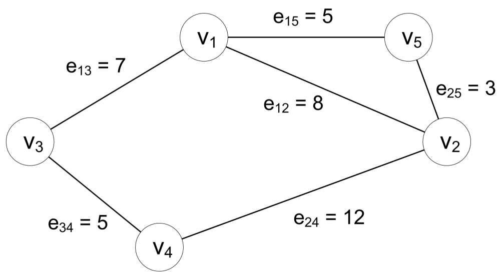


Figure 2.2: Example of a DTN Graph

2.1.2 Routing with predictable contacts

Many ideas from graph theory and network flow problems can be applied to routing in DTN [12], in general the goal is to optimize some metric (e.g. average path cost) while abiding to given constraints (e.g. link/buffer capacities).
The minimum cost metric, for example, assigns to each edge in E a link weight $w(e_i, t)$, assessing the messages arrival time $t_1 = t_0 + w(e_i, t_0)$ at the other end of an edge e_i, given the time t_0 the message is scheduled for delivery. The delay $\Delta t = t_1 - t_0 = w(e_i, t_0)$ arises from a combination of transmission-, propagation- and queuing-delays (where queuing-delays include waiting times for both local- as well as remote-queues to drain).

The *minimum expected delay* (MED) [12] algorithm seeks to minimize the average path delay in end-to-end links, with the downsides of ignoring good network links and a completely blind eye towards network congestion. A modification of the Dijkstra algorithm, as proposed in [12], addresses these drawbacks by using time-varying link weights. The time variation influences the edges capacity $c_i(t) = C(e_i, t)$, its propagation delay $d_i(t) = D(e_i, t)$ and takes into account the queue backlog $q_i(t) = Q(e_i, t)$ for edge e_i - i.e. the number of messages that are waiting to be sent. So for a message of size m, and assuming a nonzero link capacity

$C(e_i, t) = c_i(t) > 0$, the link weight is

$$w(e_i, t) = \frac{Q(e_i, t)}{C(e_i, t)} + D(e_i, t) = \frac{q_i(t)}{c_i(t)} + d_i(t) \tag{2.3}$$

The *minimum estimated expected delay* (MEED) [12] refines the scheme further by keeping a history of past contacts in order to maintain running averages on the time varying values in equation 2.3. Whenever a contact changes significantly in comparison to its running average, the network is flooded with update packets to discover the new topology. The threshold for a flooding and the number of history records used to average, determine the networks ability to react in time on topology changes. A balance between slow reaction time on the one hand, and oscillations due to too many updates on the other hand, must be found to make the network resilient.

2.1.3 Routing with opportunistic contacts

Dynamic network flows in mobile scenarios present rather difficult problems in general. Different degrees in mobility govern these scenarios, these degrees can be categorized as follow:

1. No mobile entities - Contacts appear or disappear solely based on the quality of a communication channel between them. This is the class of intermittent connectivity networks.

2. A minority of mobile entities - The few mobile nodes are exploited for their mobility. Since they are the primary source of transitive communication between two non-neighboring nodes in the network, an important routing question is how to properly distribute data among these nodes.

3. A majority of mobile entities - In this case, a routing protocol will most likely have several options available to relay a message, and may choose on how aggressively it tries to forward packets.

In the case of few mobile network entities, these are sometimes referred to as data MULES[13][14] or message ferries, a routing protocol must make use of the mules ability to carry traffic between segregated parts of the network.
Assuming that all nodes are static, have unlimited resources, and there is only one ferry available that travels on a certain path L of length $|L|$, with speed v between the segregated nodes i and j, the ferry's cycle is $T = \frac{|L|}{v}$. In one cycle the mule is able to relay data worth the size of b_{ij} with a delay of d_{ji}^L, therefore the average delay for the scenario is

$$d^L = \frac{\sum_{i,j} b_{ij} d_{ij}^L}{\sum_{i,j} b_{ij}}. \tag{2.4}$$

Any appropriate solution to the mules trajectory L needs to find an optimal tradeoff between the average delay d^L, the resulting bandwidths b^L_{ij} and a path that leads past as many nodes as possible.

The *traveling salesman problem* (TSP), which belongs to the class of NP-complete problems, addresses this process in detail. Given a (connected) weighted graph it tries to find a path that visits all nodes exactly once and has a minimum cost. As an alternative to a brute force solution, which would try all permutations and has complexity $O(n!)$, the cutting-plane method [15] offers a more efficient way to solve the set of equations by using *linear programming* with *relaxed criteria*. The relaxation transforms integer constraints into real number constraints, e.g. $x \in \{0, 1\}$ becomes $0 \le x \le 1$, and allows the algorithm to find an optimal solution to the set of inequalities. In iterative steps the algorithm then seeks a closest match between an integer valued solution and the optimal one, by comparing different integer values that are close to the optimal solution. There are several approaches to designing ferry trajectories with multiple ferries, among these are:

- Single-Route Algorithm (SIRA) [16]
- Multiple-Route Algorithm (MURA) [17]
- Node Relay Algorithm (NRA) [17]
- Ferry Relay Algorithm (FRA) [17]

In many cases, however, the path of a mule will be nondeterministic, therefore a message has to be replicated to several mules to increase the chance of a delivery. This kind of method is referred to as *epidemic routing*, the number of copies simultaneously stored in the network determines the level of *aggressiveness* the protocol uses.
For example a node may choose to give a copy to every other node it encounters, basically flooding the network, to maximize the chance of delivery and to lower the delivery time. Too much redundancy is in the best case just wasteful but it may have disastrous impact on the networks performance, congesting both the physical link and the storage capacity of network nodes.
The simplest approach to limit the number of messages simultaneously available in the network is to use randomized flooding. A message is given to a neighbor with a probability of $p \le 1$, where $p = 1$ relates to epidemic routing and $p = 0$ to using a direct transmission with no replication involved. Alternatively a scheme could be employed where the total number of copies of a message is limited, either by a fixed value or some kind of self-limiting mechanism, as in self-limiting epidemic (SLEF) [18].
Assuming that the network uses a epidemic 2-hop method to deliver messages, which means that the source node is the only replicator of the message, and all other nodes must either directly deliver the message to the destination, or eventually discard it. If there are N nodes, and if these nodes follow paths that are *independent and*

identically-distributed (i.i.d.), the expected number of transmissions per message is $E = (N-1)/2$.
The spray-and-wait protocol [19] enforces that the maximum allowable number of messages does not exceed L copies, and that a message is replicated, or 'sprayed', to L distinct relays. In the *vanilla version* of the protocol the source transmits the message to the first L nodes it encounters, whereas the *binary version* first distributes $floor(L/2)$ to the first node it encounters, and each node then transfers half of the total number of copies it has in store to any encountered node, that has no copy of the message, until there is only one message remaining in the store. The protocol then enters the wait phase, where a ferrying node will only transmit the message directly to the recipient.

2.1.4 Routing Latency

The theoretical analysis of the routing latency provides a rough estimate on a models expected performance. The *routing latency* T_i of a DTN node i is defined as the time delay between the moment a packet is generated at the sending node, and the moment it is detected in a nearby receiving node.
The *meeting (contact) time* is defined as the time until two nodes, starting form a stationary distribution, come into communication range, whereas the *inter-meeting (inter-contact) time* describes the time until two nodes, that are about to leave communication range, come into communication range once again.

The average expected delay for epidemic routing for a network with N nodes, a communication range of K and i.i.d. mobility is

$$E_D = \frac{1}{N-1}\sum_{K=1}^{N}\sum_{i=1}^{K} T_i. \tag{2.5}$$

To be able to estimate the latency T_i of a certain node, the random variable $M_{i,j}$ helps by modelling the meeting time between nodes i and j, thus

$$T_i = \min_j \{M_{i,j}\}, \textit{ and} \tag{2.6}$$

$$P(T_i < t) = P(M_{i,1} < t \textit{ or } M_{i,2} < t \ldots \textit{ or } M_{i,N-1} < t) \tag{2.7}$$

A possible assumption, motivated by considerations in chapter 2.1.5, is that the random variable $M_{i,j}$ is exponentially distributed, then the expected meeting time between i and j is

$$E[M_{i,j}] = \frac{1}{\lambda_{i,j}}, \textit{ and} \tag{2.8}$$

$$P(M_{i,j} > t) = e^{-\lambda_{i,j}t}. \tag{2.9}$$

If all nodes N behave in the same way, i.e. $\lambda = \lambda_{i,j}\ \forall i,j$, then T_i is also exponentially distributed and the expected latency becomes

$$E[T_i] = \frac{E[M_{i,j}]}{N-1} = E[T]\ \forall i,j\ and \tag{2.10}$$

$$E_D = E[T]\frac{H_{N-1}}{N-1},\ where \tag{2.11}$$

$$H_{N-1} = \sum_{i=1}^{N-1}\frac{1}{i} \tag{2.12}$$

is the harmonic sum.

Alternatively the meeting probabilities could be modeled as Markov chains, as depicted in figure 2.3, where the probability to replicate the message or to deliver it directly to its destination are both taking into account the number of messages already replicated. The node v_d is the destination node, and the nodes v_1 to v_N represent the mules, that carry the message until they encounter the destination v_D. The more nodes carry the message the higher is the probability of a direct delivery E_D and the less likely is a further replication E_R.

$$i \rightarrow i+1 : E_R[T_i] = \lambda(N-i) * i,\ \ and \tag{2.13}$$

$$E_D[T_i] = \lambda * i \tag{2.14}$$

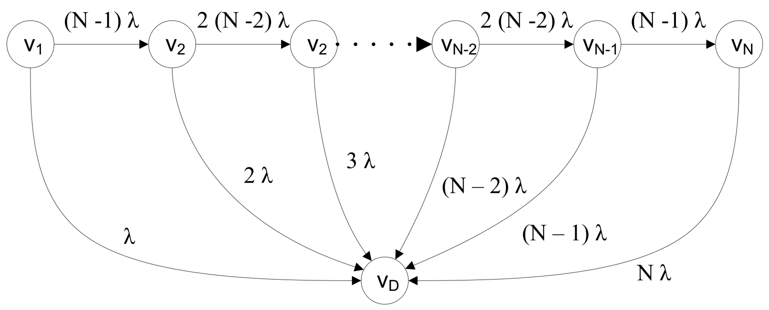


Figure 2.3: Routing Latency modelled as Markov chain

2.1.5 Predicting Future Encounters

So far all routing schemes were random, they made no assumptions what so ever about any particular network nodes - i.e. all relays are equally fast, equally capable and have similar mobility. In real life, nodes do have different capabilities (e.g. sensor, PDA, laptop) and they also show differences in movement patterns. Therefore a routing protocol may improve, if it learns to read and use these patterns to its advantage.

Statistics Based Prediction

One way of predicting future encounters is based on past encounter statistics. The *probabilistic routing protocol using history of encounters and transitivity* (PRoPHET) [20] uses an algorithm that attempts to exploit the non-randomness of real-world encounters by maintaining a set of probabilities for successful delivery to known destinations in the delay-tolerant network (delivery predictabilities), and by replicating messages during opportunistic encounters, but only if the mule that does not have the message, appears to have a better chance of delivering it. Every mule M stores delivery predictabilities $P(M,D)$ for each known destination D, if a predictability value is unknown its assumed to be zero. The delivery predictabilities used by each mule are recalculated at each opportunistic encounter according to three rules:

1. When the mule M encounters another mule E, the predictability for E is increased using the initialization constant $L_{encounter}$,

$$P(M,E)_{new} = P(M,E)_{old} + (1 - P(M,E)_{old}) * L_{encounter}. \tag{2.15}$$

2. The predictabilities for all destinations D other than E are 'aged' by γ^K, where γ is the aging constant and K is the number of time units that has elapsed since the last aging:

$$P(M,D)_{new} = P(M,D)_{old} * \gamma^K \tag{2.16}$$

3. Predictabilities for other nodes in the network are exchanged between M and E, and the 'transitive' property of predictability is used to update the predictability for a destination D, for which E has a $P(E,D)$ value, on the assumption that M is likely to meet E again, using a constant scaling factor β:

$$P(M,D)_{new} = P(M,D)_{old} + (1 - P(M,D)_{old}) * P(M,E) * P(E,D) * \beta \tag{2.17}$$

Mobility Profile Based Prediction

Another approach is to use model based prediction, as proposed by Becker et al. [21], where an abstract mobility model serves as basis and past encounters trim the models parameters to predict future encounters. A mobility-profile based prediction scheme separates the network into K locations, and represents each network node as a K-dimensional vector $\vec{M}_n$ containing the sojourning probabilities for all locations. By using e.g. Euclidean distance between two nodes m and n as the metric

$$|\vec{M}_n - \vec{M}_m| = \sqrt{\sum_{i=1 \ldots K} (M_n(i) - M_m(i))^2} \tag{2.18}$$

It is then possible to use this metric as a basis to predict the encounter probability $P(m,n) = f(|\vec{M}_n - \vec{M}_m|)$ for two nodes. The model must carefully chose the parameter K to avoid either too little overlap or too crude resolution, and each node must be able to learn the sojourning probabilities for its vector $\vec{M}_n$, either by using wireless network *access points* (APs) locations or the *global positioning system* (GPS) or a combination thereof to obtain

$$M_n(i) = \frac{Time\ at\ location\ i}{Timewindow}. \tag{2.19}$$

The $Timewindow$ variable in equation 2.19 represents the total time, i.e. the sum of the times for every location i.

Social Profile Based Prediction

To relate social networks to DTN, one has to be aware that in a mobile scenario nodes are driven by humans, which have social relations that govern their mobility patterns. Instead of having a routing scheme that operates on set of link weights defined by technical parameters, the link weights could model social relations between nodes and furthermore be able to identify node communities. SimBet [22] is an example of such a protocol.

Research in the field of social networks dates back to the early last century, where scientists sought an answer to the shape and the basic properties of social networks. A fascinating experiment, called the small world experiment, and devised by Milgram et al. [23], took place in the USA in 1967 and revealed astonishing results.
The experiment selected arbitrary ”starting persons” who where asked to forward a letter to another randomly selected person living in a town separated by a large geographical distance. Each participant was asked to exploit his social network to deliver the letter, furthermore each person forwarding the letter was requested to send a postcard to the researchers, providing them with a means to track a letters route on its way to the destination. A majority of the letters did not make it to the destination, but those who did showed, that the average path length was around six persons, therefore the term 'six degrees of separation' was coined.
Later studies examined other networks, like the internet topology or cellular networks in biology, where interactions between the cell's numerous constituents, such as proteins, DNA, RNA and small molecules are modelled as a network. These newer studies confirmed the results of Milgram's experiment, and showed that there are several common characteristics that apply to these kinds of networks.

Social networks belong to the type of *small-world networks*, these networks have in common that its nodes form clusters, or groups, and that certain nodes provide links to other clusters or groups, interconnecting these subnetworks. The major properties of social networks are, that they are Scalefree [24] and that they possess a high clustering coefficient c_i and a small path length [25]. Actually there are three classes of scale-free networks: (a) real scale-free networks, characterized by a vertex connectivity distribution that decays as a power law; (b) broad-scale networks, characterized by a connectivity distribution that has a power law regime followed by a sharp cutoff; and (c) single-scale networks, characterized by a connectivity distribution with a fast decaying tail, according to Amaral et al [26]. In general this means that there are few nodes having many connections to other nodes, these nodes are called hubs, and the vast majority of remaining nodes have few connections to other nodes. An example of such a network is shown in figure 2.4. Besides the already addressed average path length there are other properties that help classifying these networks.

- Path length, number of hops between two nodes on the average,

$$l = \frac{1}{n(n-1)/2} \sum_{i>j} d_{ij}. \tag{2.20}$$

- Degree distribution, the probability p(k) of a node having k neighbors,

$$p(k) \propto k^{-\gamma} \tag{2.21}$$

- Clustering coefficient C_i quantifies how close the vertex and its neighbors are to being a clique (complete graph). If links A-B and B-C exist, the clustering coefficient predicts the probability that A-C exists.

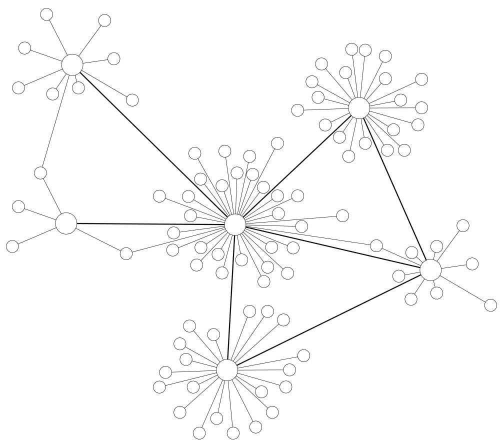


Figure 2.4: Graph of a scale-free network

2.1.6 Content Dissemination

In contrast to the routing schemes described so far in this chapter, where a message or packet was given the primary focus, it may be interesting to investigate the

dissemination of content as a whole. The general assumption is, that there might be information that is valuable to more than one network entity, which is also the basic assumption of the PodNet application.
Each network node has to be aware of available content and eventually synchronize with elected nodes, if content interests match. Markov chains were already addressed in chapter 2.1.4, and can also be used to model content dissemination as in [5], the expected delay if no cooperation strategy is employed is

$$E_D = \sum_{i=1}^{N-1} \frac{1}{\lambda_i} \xrightarrow{\lambda = const.} E_D = \frac{1}{\lambda} \sum_{i=1}^{N-1} \frac{1}{N-i} = \frac{1}{\lambda} \sum_{j=1}^{N-1} \frac{1}{j} = \frac{1}{\lambda} H_{N-1}. \tag{2.22}$$

$$H_{N-1} = \sum_{i=1}^{N-1} \frac{1}{i}$$

If however the cooperation is unlimited, or put another way, the mutual interests in content are absolutely identical, the expected delay becomes

$$E_D = \frac{2}{\lambda N} H_{N-1} \tag{2.23}$$

Fluid models (Deterministic) take an approach inspired by biology, it is assumed that the number of nodes $N \rightarrow \infty$ is infinite, and that there is a rate of infected nodes $I(t)$, furthermore the change in the rate of infected nodes $\frac{d}{dt} I(t)$ adheres to the differential equation

$$\frac{d}{dt} I(t) = \lambda(N - I(t))\ I(t). \tag{2.24}$$

The probability of a message being delivered in time t is $P(t)$, and the probability that the destination meets one of the $I(t)$ infected nodes is $M(t)$, therefore

$$P(t) = P(t > T_D) = 1 - P(t \leq T_D),\ and \tag{2.25}$$

$$\frac{d}{dt} P(t) \propto \frac{d}{dt} M(t) * P(t \leq T_D) \propto \lambda * I(t) * (1 - P(t)) \tag{2.26}$$

The system of *ordinary differential equations* (ODEs) can then be solved,

$$I(t) = \frac{1}{1 + (N-1)e^{-\lambda N t}} \tag{2.27}$$

$$P(t) = \frac{N}{(N-1) + e^{-\lambda N t}}. \tag{2.28}$$

And the resulting expected delay is

$$E_D = \int_0^\infty (1 - P(t))dt = \frac{ln(N)}{\lambda(N-1)}. \tag{2.29}$$

2.1.7 Resource Restrictions

An important consideration in DTN is the availability of network resources. Many nodes, such as mobile phones, are limited in terms of storage space, transmission rate, and battery life. Others, such as buses on the road, may not be as limited. Routing protocols can utilize this information to best determine how messages should be transmitted and stored, to not over-burden limited resources. Only recently has the scientific community started taking resource management into consideration, and it is still a very active area of research.

2.2 ANA Framework

The *Autonomic Network Architecture* (ANA) is a network research project initiated and supported by different universities in Europe [27].
The goal is to develop a novel network architecture, an alternative to the ubiquitous TCP/IP Network Layer model, and populate it with the functionality needed to demonstrate the feasibility of autonomic networking. To avoid the pitfalls of past architectures, ANA obeys certain guiding principles, i.e. *maximum flexibility* and *functional scaling*, by design.

2.2.1 Concepts

The ANA framework introduces fundamental changes to the well-established concepts in internetworking, as it smashes the layered (stacked) approach and rather operates on a flat hierarchy of *functional blocks*, called bricks in ANA terminology, than a fixed set of stack elements.
This crucial difference introduces the possibility to dynamically arrange data flows through individual functional blocks, even rearrange them at runtime. It is thus the choice of different brick sequences that determines the functional abilities of a certain ANA entity, therewith fullfilling the criteria for both maximum flexibility and functional scaling.
The ANA framework can be roughly divided into its core process, called *minmex*, and a set of attachable plugins called *bricks*. The minmex is the primary hub for all interactions within an ANA node, it offers both a node local *inter-process communication* (IPC) facility and access to the host systems *network interface card* (NIC). Additionally the minmex receives loading instructions for bricks from the minmex configuration tool *mxconfig*, a shell program that supports scripting. Furthermore the ANA bricks make use of the frameworks *application programming interface* (API) to link with other functional blocks, thereby obtaining a desired functionality.

2.2.2 Terminology

To give a brief introduction, the most important concepts are addressed shortly, and will later be described in detail in the following sections. The ANA framework introduces its own terminology to describe the fundamental concepts of the frameworks approach [9]. First and foremost ANA differentiates between the *node compartment* and other compartments, e.g. an ethernet *network compartment*. An ANA brick may introduce a new compartment to the system, by implementing a compartment protocol, which defines a policed set of functional blocks and the data flow therein. As already mentioned the *functional block* (FB) is the atomized representation of a processing function in ANA, it generates, consumes and processes information. Each functional block runs in a certain minmex instance, therefore the *node compartment* encompasses all functional blocks that run in the *same minmex*

instance. Network compartments are of a different nature, as its participants must communicate over a physical link of some kind (e.g. Ethernet), and therefore depend on certain hardware related functional blocks.
The minmex uses *information dispatch points* (IDPs) to relay informations between bricks, the IDP serves as the key to a callback function table, called the *information dispatch table* (IDT). Before a brick is able to receive information it needs to publish an IDP together with a list of keywords to the *key-value repository* (KVR) of its minmex instance, additionally it needs to register a callback function with the IDT where it will receive data for the published IDP. The keywords published in the KVR serve as service descriptors, so bricks may find each other. Figure 2.5 summarizes the addressed properties and their relations.

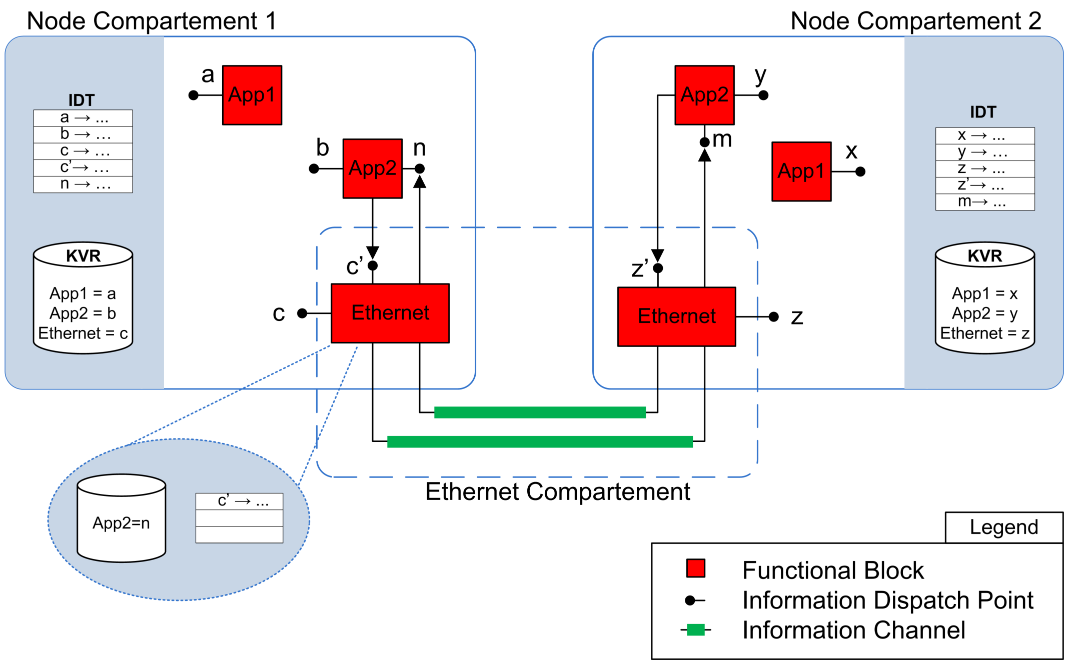


Figure 2.5: Schematic view of an ANA Compartment

2.2.3 Functional Blocks

The core building block of the ANA framework is the *functional block* (FB), and as already stated, FBs are code instances that can process (send, receive, forward, etc.) data. FBs can be composed of one, or a set of several FBs. The fact that a FB can represent the whole range from an individual processing function (as an atomic FB) to a whole compartment stack or even a network node (as a composed FB), makes this abstraction very useful.

2.2.4 Compartments

A compartment represents an abstraction that allows decomposition of communication systems and networks into smaller and more easily manageable units. Such an atomization is motivated by recent research[28], which led to the conclusion, that today's networks have to deal with such a large diversity of commercial, social and governmental interests, that a pragmatic way to resolve these tussles, is to logically divide the network into different realms[29] or turfs[30]. Additionally to the network partitioning, the compartment abstraction also serves as a basic unit for the federation of compartments into global-scale communication systems and networks.
As already mentioned compartments do implement the operational rules and administrative policies for a given communication context, thereby also defining the technological and/or administrative boundaries for a given context. It is worth noting that compartments have full autonomy on how to handle the compartment's internal communication - i.e. there are no global invariants that have to be implemented by all compartments or every communication element.
There are however requirements, that each compartment has to fullfill, in order to integrate properly into the framework. Among these conditions are:

- **Registration and Deregistration**: Compartments require some kind of registration or publish function that allows communication entities to become a member of a compartment, if they wish so. Likewise a deregistration or unpublish function is needed, to signal a leave to the compartment.
- **Policy Enforcement**: Compartments can police their members and resources, e.g. a compartment may enforce that all its functional blocks require a proper authentication and/or data encryption.
- **Identifier Management**: Most compartments will make use of some kind of naming or addressing scheme to identify individuals or group members within the compartment. Typically a (pseudo) unique, compartment-local identifier will serve this purpose. Managing these identifiers, and their uniqueness, rests with the compartment provider.
- **Identifier Resolution**: Of course the compartment provides a way to access individual members or groups that already obtained an identifier for themselves. The result of this identifier resolution process depends on the type of compartment and how communication within the compartment is handled, but basically the result is another identifier that is required for communication on a lower level.

2.2.5 Information Channel

Communication inside a compartment is mediated via *information channels* (ICs), which can be of either physical or logical nature. Examples of physical ICs are a wired link, a wireless medium or the local memory, while logical (or virtual) ICs are represented by a chain of packet processing elements and further ICs. The information channel abstraction is able to capture various types of communication channels, ranging from point-to-point links or connections, over multicast or broadcast trees to special types of channels as anycast or concast trees.

2.2.6 Information Dispatch Point

In order to flexibly connect to a FB or IC, the ANA framework introduces *information dispatch points* (IDPs), which can be dynamically bound and hence represent a decoupled “entry point” or handle for each FB or IC. The fact that this decoupling occurs in a transparent way for the involved entities, relieves these entities of being aware of, or even being involved in, any (autonomic) re-binding procedure that can take place during active communications. A clear distinction between IDPs and FBs may be difficult sometimes, as it can be argued, that IDPs are just a special type of FBs. In contrast to FBs however, IDPs are limited to perform only data forwarding operations (based on their bindings).

2.2.7 Information Dispatch Table

Every IDP is bound to a certain FB or IC, in that way that every entity publishing an IDP will associate an (internal) information retrieval function for that IDP. Typically this is done by the means of a callback function associated to an IDP. The minmex keeps track of every IDP and its callback function in its *information dispatch table* (IDT).

2.2.8 Key-Value Repository

The compartment prerequisites state, that each compartment must take care of its identifier space, naturally this also applies to the node compartment. The minmex keeps track of published identifiers in its *key-value repository* (KVR), and answers identifier resolution request by searching the KVR. An entry to the KVR basically relates a keyword to an IDP.

2.2.9 Minmex

The *minmex controller* (MC) is a truly autonomic process, it runs dynamically loadable bricks and performs a continuous assessment of the basic operation of an ANA node, i.e. a sanity and health check of the components running inside the node. The core objective of the MC is in fact to protect running elements from faulty or misbehaving elements, in order to guarantee the performance of the ANA subsystem. Among the tasks are:

- Periodical control of the data forwarding paths in the IDT, in particular detection and removal of accidentally created loops.

- Garbage collection of unused or expired IDPs, and IDPs that became an orphan due to an entity malfunctioning or crashing.
- Deletion or re-instantiation of malfunctioning or crashed functional blocks within the system.
- Support for several IDTs, including the management of access rights for IDTs.

2.2.10 API

The frameworks *application programming interface* (API), also referred to as 'compartment API' in other documents, is the pivot for every functional block, as it offers interfaces to communicate with both the minmex and other currently loaded bricks. The API can roughly be divided into its core functionality and a set of auxiliary functions. The core itself comes in three layers, the differences in these layers are best summarized as a tradeoff between flexibility versus comfort. The numerous auxiliary functions in the API cover many subjects, among these are:

- Platform independent support for threads, timers and mutual exclusion mechanisms.
- Storage container support (e.g. simple linked lists, hashed tables, etc.).
- Inter-process communication support through a standardized message format called XRP.

Besides the API itself the ANA framework also offers brick templates, these contain necessary API inclusions and provide a basic brick-skeleton - i.e. the minimum set of prerequisites each brick must meet.

Core API

As already mentioned there are three layers to the core API, each having different features. At the lowest level there is the `anaL0` API set, which encompasses the functionality needed to meet the lowest common denominator for a basic system operability. This level includes functions for sending data to an IDP, publishing an IDP and associate a callback function for subsequent data retrieval. The `anaL0` does not specify any particular pattern for messages passed through the system, the support for XRP formatted messages is only made available by the `anaL1` API set. The new API level adds XRP message encoding- and decoding-functions, these messages implement the compartment requirements imposed by the compartment specifications. Having the properly formatted XRP messages and the service that can relay them, the specifications are met, beyond this level the `anaL2` API set merely introduces comfort. Basically the `anaL2` API level consist of wrapper functions summarizing all the individual steps needed to set-up IDPs with associated callback functions in order to implement a compartment protocol.

2.3 PodNet

As an example of a DTN application, and also as the basis for this thesis, the *wireless ad-hoc podcasting network* (PodNet) application is providing everything necessary to operate in a mobile, delay-tolerant environment. PodNet [2] was devised at the ETH Zurich in 2007 and is based on research, done in the field of mobile networking, by the communication system group (CSG).

Podcasting has become a popular service lately, as it allows individuals to remain up to date on recently published informations without their interaction needed. Podcasting describes the process where a content publisher offers a podcasting channel to the public, by e.g. publishing it on the Internet, where individuals have the possibility to subscribe to the podcast. Later on, everytime the publisher adds a new entry to his podcast, each subscriber will periodically check and eventually discover the update and download the new content automatically.
PodNet enhances the classical podcasting service by extending the publish/subscriber scheme to every node, this allows every individual to become both a publisher of and a subscriber to a podcasting channel. The application is designed to be operated in a mobile scenario, it uses the ad-hoc mode for wireless 802.11a/b/g networks as its physical link and TCP/IP for addressing and transmission control. PodNet uses opportunistic contacts to propagate data from mobile to mobile. If two devices are within transmission range, the common subscribed channels are synchronized.

2.3.1 Structure

The PodNet application is written in object-oriented C++, as a collection of threads that fullfill different tasks in parallel and communicate with each other by using messages stored in a common memory region. The modules that make up the PodNet application are depicted in figure 2.6, the graph shows the message flow between individual tasks or modules and the separation into system threads.

A detailed analysis of [2] reveals, that the ’Router’ and ’Sync’ module together provide the DTN scheme. By broadcasting discovery packets and keeping track of the packets received from other peers, they enable the podcasting application to identify and to contact potential candidates for content exchange. The ’Transfer Server’ and ’Transfer Client’ modules do implement the podcasting protocol, that is required for actual information exchange. Besides these modules the application offers interfaces to podcasting data, event logging and live traffic analysis. The interaction between user and application is facilitated either by a textual or a graphical *user interface* (UI), depending on the users choice.

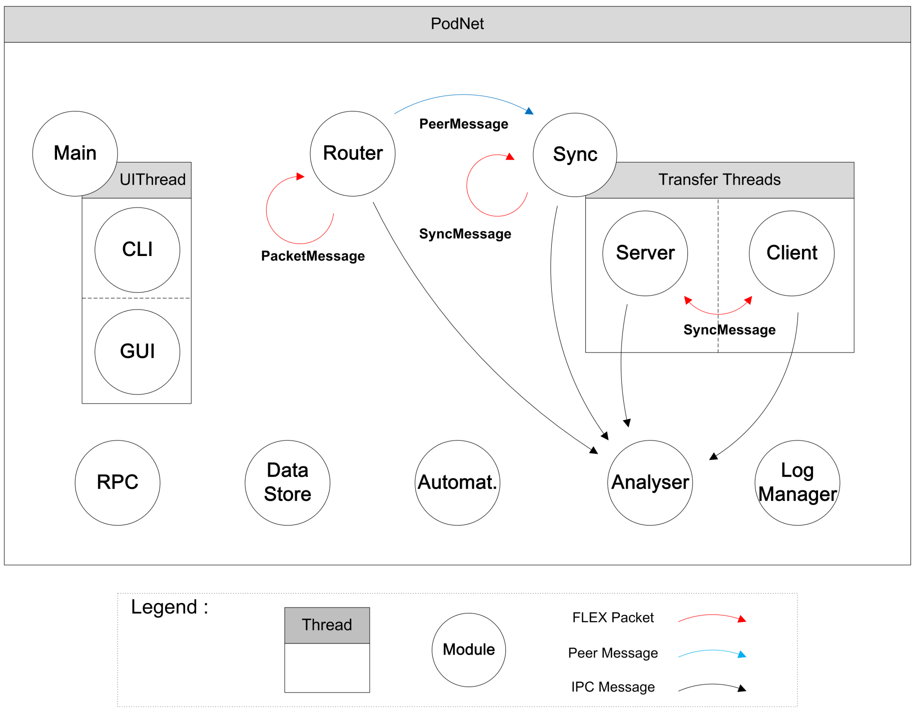


Figure 2.6: Schematic view of a PodNet entity

2.3.2 Discovery and Synchronization

Every network node sends a periodic message to all neighbors it can reach, informing them of its presence. By using the networks broadcast address it is able to spread a discovery packet, containing vital information for potential candidates, to a number of network nodes at once. As each node observes the broadcasted discovery packets, it is not only able to build a network graph as introduced in chapter 2.1, but also to estimate a connections quality by keeping track of overdue packets. Once the set of neighbors is known, it is a strategic question of how to start synchronizing with every neighbor, e.g. immediately after its discovery or after some delay to reduce runs on new peers. Further analysis in [2] also showed, that the amount of the initially transferred data volume, between two nodes, affects the fairness of the dissemination. Unlimited transfer volumes can stall the spreading process considerably. So the protocol must restrain from large bulk transfers, and rather seek to spread information evenly over all interested nodes.

2.3.3 Content Exchange

In allusion to popular podcasting protocols PodNet describes a podcast as a set of *channels*. Every channel has a unique identifier and contains a variable number of *episodes*, each episode is uniquely identifyable too and represents an atomic information unit. Usually an episode is made up of an authors name, an episodes title, a date and a variable number of data blocks, e.g. file data. To improve the performance when searching for mutual interests, expressed as subscriptions to a unique channel identifier, PodNet makes use of bloom-filters [31] that reduce the complexity of the searching problem from the exponential to the linear time domain.

2.3.4 Network Message Format

Both the discovery and the content exchange protocol in PodNet make use of the FLEX [2] message format, a message encoding/decoding scheme derived from the OBEX [32] format. Figure 2.7 shows the different protocol encapsulations for a typical FLEX formated message on an Ethernet link. The FLEX format takes care of platform dependent byte ordering and facilitates data splitting for messages larger than a given *maximum transmission unit* (MTU) size.

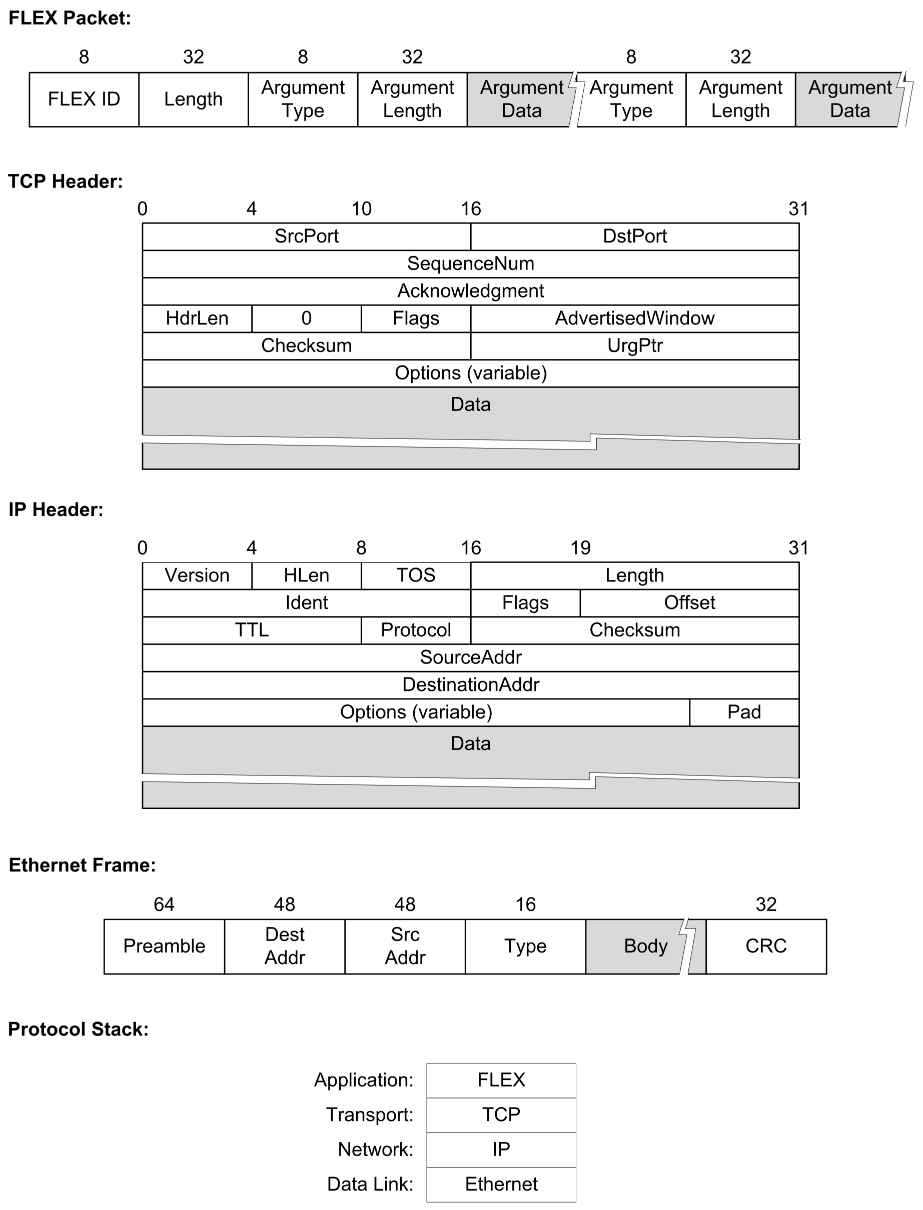


Figure 2.7: Typical network Protocol encapsulation of a PodNet message

2.4 Transmission Control Protocol

In a mobile network scenario the loss or corruption of packets during transit is not uncommon, furthermore wireless links are more susceptible to congestion than wired networks, as the wireless link, being a classical ethernet network, is a shared medium. Therefore a link-level protocol that wants to deliver frames reliably must somehow recover from these discarded or lost frames. The *transmission control*

protocol (TCP) [10] is a well established and flexible protocol, it proved its reliable transmission capabilities in a multitude of networking scenarios, and is considered an excellent example of a solution to the problem of reliable transmission. Currently the ANA framework does not offer a comparable service, therefore TCP is implemented in the framework, as a part of thesis, following the concepts introduced in this chapter.
This section is largely based on the explanations in "Computer Networks" written by Larry Peterson and Bruce Davie [33].

2.4.1 Error Recovery

The recovery from a frame that was lost during transmission is usually accomplished using a combination of two fundamental mechanism - *acknowledgements* (ACK) and *timeouts*. An acknowledgement is a small control frame that the protocol sends back to its peer informing it on the successful reception of an earlier frame. In the case of TCP the control frame is a complete TCP header with no payload, the layout of a TCP header has been introduced in figure 2.7. If the sender does not receive an acknowledgement after a reasonable amount of time, i.e. after a timeout occurred, it must assume that a frame got lost and therefore *retransmit* the original frame. The general strategy of using acknowledgements and timeouts to implement reliable delivery is sometimes called *automatic repeat request* (ARQ).
The simplest ARQ scheme is the *stop-and-wait* algorithm. The idea of stop-and-wait is that after transmitting one frame, the sender has to wait for an acknowledgement before it can transmit the next frame. If the acknowledgement does not arrive after a certain period, the sender times out and retransmits the original frame. The main shortcoming of the stop-and-wait algorithm is that it allows the sender to have only one outstanding frame on the link at a time, and this may be far below the link's capacity. The following consideration illustrates this situation: A 1.5-Mbps link with a 45-ms *round-trip time* (RTT) has link capacity of $capacity = delay \times bandwidth \approx 8kByte$, since the sender can send only one frame per RTT, and assuming that a frame has the size of 1kB, the link utilization is only 12.5%. To use the link fully, then, the sender should be able to transmit up to eight frames before having to wait for an acknowledgement.

2.4.2 Sliding Window Algorithm

At the heart of TCP lies the sliding-window algorithm, it ensures data continuity and consistent memory space for packet processing. The algorithm models a circular data buffer and maintains a set of pointers to items of special significance. TCP actually use two such buffers, one for the receiving side and one for the sending side. An illustration of the two circular buffers used in TCP is given in figure 2.8.

First, the sender assigns a *sequence number* (SeqNum) to each frame that is scheduled for delivery. For now, the fact that the SeqNum will be implemented by a finite-size variable shall be ignored, and instead it shall be assumed that SeqNum

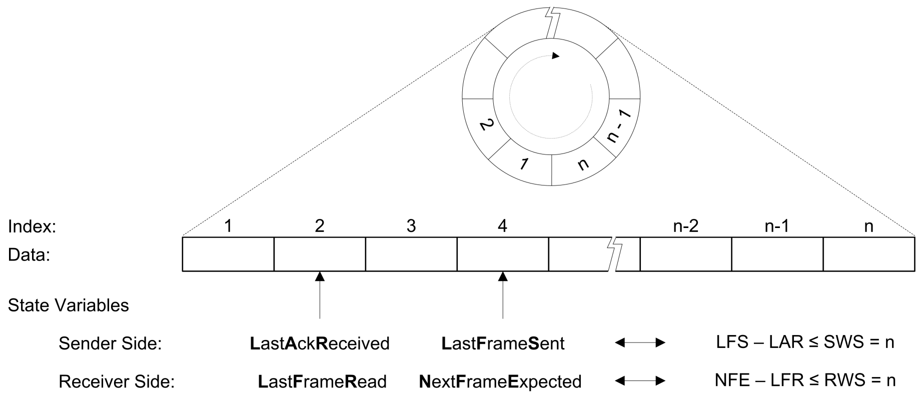


Figure 2.8: Sliding-window - Circular data buffers in TCP

can grow infinitely large.

The sliding window algorithm on the sending side works as follows. The sender maintains three variables: The *send window size* (SWS), which gives the upper bound on the number of outstanding (unacknowledged) frames that the sender is allowed to transmit; LAR denotes the sequence number of the *last acknowledgement received*; and LFS denotes the sequence number of the *last frame sent*. The sender enforces that $LFS - LAR \leq SWS$ in order to ensure data consistency. When an acknowledgement arrives the sender is allowed to progress its LAR pointer, thereby allowing a new frame to be sent. Also, the sender associates a timer with each frame it transmits, and it retransmits the frame should the timer expire before an ACK is received. Note that the sender has to be willing to buffer up to SWS frames since it must be prepared to retransmit them until they are acknowledged.

The receiver side has to maintain the following three variables: The *receive window size* (RWS), giving an upper bound on the number of out-of-order frames that the receiver is willing to accept; LAF denotes the sequence number of the *largest acceptable frame*; and LFR denotes the sequence number of the *last frame read* by the receiving application. The receiver maintains the invariant $LAF - LFR \leq RWS$ to maintain data consistency. When a frame with sequence number $SeqNum$ arrives, the receiver takes the following action. If $SeqNum \leq LFR$ or $SeqNum > LAF$, the the frame is outside the receiver's window and it is discarded. If $LFR < SeqNum \leq LAF$, then the frame is within the receiver's window and it is accepted. Now the receiver needs to decide whether or not to send an ACK. Let $SeqNumToAck$ denote the largest sequence number not yet acknowledged, such that all frames with sequence numbers less than or equal to $SeqNumToAck$ have been received. The receiver acknowledges receipt of $SeqNumToAck$, even if higher-numbered packets have been received, such an acknowledgement is said to be cumulative. The receiver then sets $LFR = SeqNumToAck$ and adjusts $LAF = LFR + RWS$.

2.4.3 Finite State Machine

The well-designed nature of TCP allows an elegant implementation of the protocol, by using the model of a *finite state machine* (FSM). A FSM is a graph G with a set of nodes V that are interconnected by a set of directed edges E. The nodes V denote the different states of the state machine, whereas the edges E model a state transition of one state into another

$$e_{12}: \quad v_1 \xrightarrow{c=1/a=2} v_2, \tag{2.30}$$

where the condition $c = 1$ must be met in order to follow an edge into a new state, causing the action $a = 2$ to be executed.. The FSM diagram for TCP is shown in figure 2.9, where the criteria for state transitions are based on the kind of packet received, TCP uses flags in the packet header to differentiate different packet types.

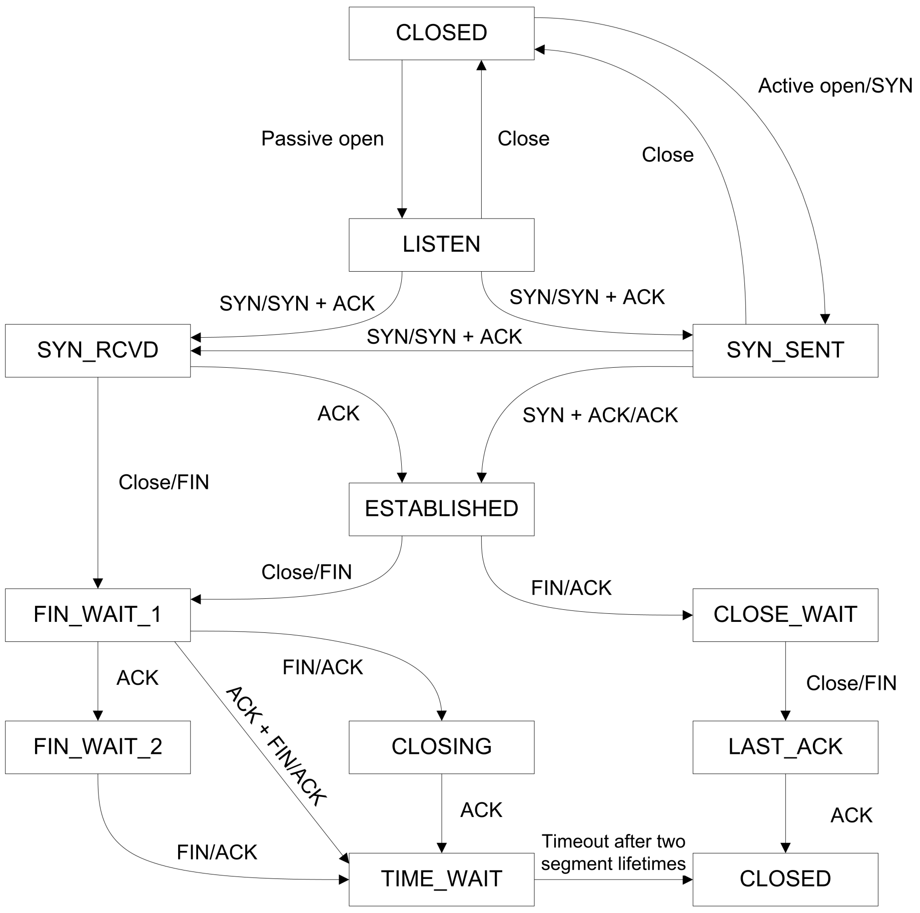


Figure 2.9: TCP modeled as a finite state machine

2.4.4 Other Features

There exist many implementations of the transmission control protocol, due to proposed improvements being applied and different interpretation of the specifications. The following listing provides an overview of features inherent in most TCP implementations.

Three-Way Handshake

The idea is that two peers need to agree on a set of parameters, which, in the case of opening a TCP connection, are the starting sequence numbers the two sides plan to use for their respective data frames. First the client (the active participant) sends a segment to the server (the passive participant) stating the initial sequence number it plans to use (Flags = `SYN`, SeqNum = x, see the respective fields in figure 2.7). The server then responds with a single segment that both acknowledges the client's sequence number (Flags = `ACK`, AckNum = $x + 1$) and states its own beginning sequence number (Flags = `SYN`, SeqNum = y). That is, both the `SYN` and `ACK` bits are set in the `Flags` field of this second message. Finally, the client responds with a third segment that acknowledges the server's sequence number (Flags = `ACK`, AckNum = $y+1$). The reason that each side acknowledges a sequence number that is one larger than the one sent is that the `Acknowledgement` field actually identifies the 'next sequence number expected', thereby implicitly acknowledging all earlier sequence numbers. The three-way handshake is illustrated in figure 2.10.

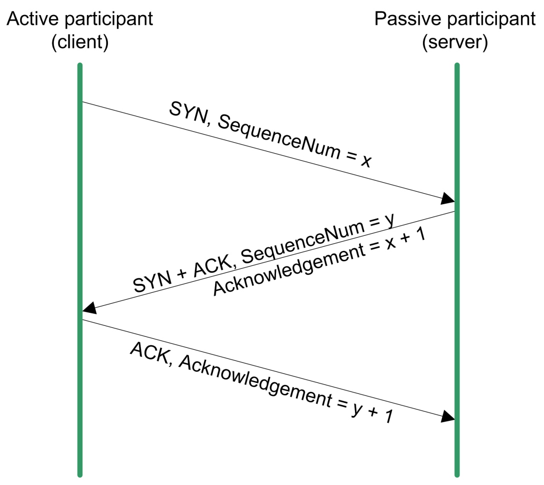


Figure 2.10: TCP - Three-Way Handshake

Flow Control

In order to incorporate flow control between the sender and the receiver, TCP folds the flow-control function into the sliding window algorithm. In particular, rather than having a fixed-size sliding window, the receiver advertises a window size to the sender. This is done using the `AdvertsiedWindow` field in the TCP header (see figure 2.7). The sender is then limited to having no more than a value of `AdvertisedWindow` of unacknowledged data at any given time. The receiver selects a suitable value for `AdvertisedWindow` based on the amount of memory allocated to the connection for the purpose of buffering data. The idea is to keep the sender from overrunning the receiver's buffer. On the receiver side this introduces some changes, additionally to keeping track of the *next frame expected* (NFE) it must track the *last frame read* (LFR) by the application, receiving the data stream, and finally adapt its constraint on the receiver side, that avoid overflowing its buffer

$$(NFE - 1) - LFR \leq RWS. \tag{2.31}$$

Therefore the peer advertises a window size of

$$AdvertisedWindow = RWS - ((NFE - 1) - LFR), \tag{2.32}$$

which represents the amount of free space remaining in its buffer. As data arrives, the receiver acknowledges it as long as all the preceding frames have also arrived. In addition, NFE is increased, meaning that the advertised window potentially shrinks. Whether or not it shrinks depends on how fast the local application process is consuming data. If a `AdvertisedWindow` value should ever reach 0, TCP on the send side must then adhere to the advertised window it gets from the receiver, and stop sending data. This means that it must ensure that

$$LFS - LAR \leq AdvertisedWindow. \tag{2.33}$$

Said another way, the sender computes an effective window that limits how much data it can send

$$EffectiveWindow = AdvertisedWindow - (LFS - LAR). \tag{2.34}$$

An advertised window of 0 means that the sending side cannot transmit any data, even though data it has previously sent has been successfully acknowledged. Finnally, not being able to transmit any data means that the send buffer fills up, which ultimately causes TCP to block the sending process. As the receiver may eventually start draining its buffer, the sender has no knowledge of this, because it is not permitted to send any more data that could trigger an acknowledgement containing the new `AdvertisedWindow` value. TCP on the receiver side does not spontaneously send nondata segments, it only sends them in response to an arriving data segment. TCP deals with this situation as follows. Whenever the other side advertises a window size of 0, the sending side persist in sending a segment with one byte of data every so often. It knows that this data will probably not be accepted, but it tries anyway, because each of these 1-byte segments will trigger a response that contains the current advertised window. Eventually, one of these 1-byte probes triggers a response that reports a nonzero advertised window.

Adaptive Retransmission

Because TCP guarantees the reliable delivery of data, it retransmits each segment if an ACK is not received in certain period of time. TCP sets this timeout as a function of the *round trip time* (RTT) it expects between the two ends of the connection. Unfortunately, given the range of possible RTTs between any pair of hosts in a large inter-network, as well as the variation in RTT between the same two hosts over time, choosing an appropriate timeout value is not that easy. To address this problem, TCP uses an adaptive retransmission mechanism, that relies on a running average of the RTT between two hosts. Specifically, every time TCP sends a data segment, it records the time. When an ACK for that segment arrives, TCP reads the time again and then takes the difference between these two times as a $SampleRTT$. TCP then computes an $EstimatedRTT$ as a weighted average between the previous estimate and this new sample, that is,

$$EstimatedRTT = \alpha \times EstimatedRTT + (1 - \alpha) \times SampleRTT. \tag{2.35}$$

The parameter α is used to smooth the $EstimatedRTT$, the TCP specification recommends a setting of α between 0.8 and 0.9. TCP then uses $EstimatedRTT$ to calculate the timeout in a rather conservative way:

$$Timeout = 2 \times EstimatedRTT \tag{2.36}$$

An important note on the subject of sampling round trip times, which also remained hidden for quite a long time after the protocol has been introduced, is, that the RTT should only be sampled if the ACK was for a segment where no retransmissions occurred. It showed that the inability to unambiguously attribute an ACK to a certain (re-)transmission, falsified the $EstimatedRTT$ to such an extent, that it negatively influenced the performance.

Record Boundaries

Since TCP is a byte-stream protocol, the number of bytes written by the sender are not necessarily the same as the number of bytes read by the receiver. For example, the application might write 8 bytes, then 2 bytes, then 20 bytes to a TCP connection, while on the receiving side, the application reads 5 bytes at a time inside a loop that iterates periodically. TCP does not interject record boundaries between the 8^{th} and 9^{th} bytes, nor between the 10^{th} and the 11^{th} bytes. This is in contrast to a message-oriented protocol, such as the *user datagram protocol* (UDP), in which the message that is sent is exactly the same length as the message that is received.
Even tough TCP is a byte-stream protocol, it has two different features that can be used by the sender to insert record boundaries into the byte-stream, thereby informing the receiver how to break the stream of bytes into records.
The first mechanism is the urgent data feature, as implemented by the `URG` flag and the `UrgPtr` field in the TCP header (see figure 2.7). Originally, the urgent data

mechanism was designed to allow the sending application to send distinct data, i.e. separate from the normal flow of data. This out-of-band data was identified in the segment using the `UrgPtr` field and was to be delivered to the receiving process as soon as it arrived, even if that meant delivering it before data with an earlier sequence number. Over time, however, this feature has not been used, so instead of signifying “urgent” data, it has come to be used to signify “special” data, such as a record marker.
The second mechanism for inserting end-of-record markers into a byte-stream is the push operation. Originally, this mechanism was designed to allow the sending process to tell TCP that it should send (flush) whatever bytes it had collected to its peer. The push operation can be used to implement record boundaries because the specification says that TCP must send whatever data it has buffered at the source when the application says push, and optionally, TCP at the destination notifies the application whenever an incoming segment has the `PUSH` flag set. If the receiving side supports this option, the push operation can be used to break the TCP stream into records.

Congestion Control

The essential strategy of TCP to control congestion is to send packets into the network and then to react to observable events that occur. The goal is to determine for each source how much capacity is available in the network, so that it knows how many packets it can safely have in transit. Once a given source has this many packets in transit, it uses the arrival of an ACK, as a signal that one of its packets has left the network, and that it is therefore safe to insert a new packet into the network, without adding to the level of congestion. By using ACKs to pace the transmission of packets, TCP is said to be self-clocking.
TCP maintains a new state variable for each connection, called `CongestionWindow`, which is used by the source to limit how much data it is allowed to have in transit at a given time. The `CongestionWindow` is in concurrency with the `AdvertisedWindow`, introduced earlier in the section on flow control, as the smaller value of these two becomes the basis for a computation of the effectively allowed window size `EffectiveWindow`. Thus it is the `EffectiveWindow` size that dictates whether TCP is allowed to insert another packet into the network,

$$MaxWindow = min(CongestionWindow, AdvertisedWindow) \quad (2.37)$$
$$EffectiveWindow = MaxWindow - (LFS - LAR). \quad (2.38)$$

A TCP source sets the value of `CongestionWindow` based on the level of congestion it perceives to exist in the network. This involves decreasing the congestion window when the level of congestion goes up and increasing the congestion window when the level of congestion goes down. TCP uses an additive increase/multiplicative decrease scheme to alter the congestion window’s value.
Based on the observation that the main reason packets are not delivered, and a time-out results, is that a packet was dropped due to congestion. It is rare that a packet

is dropped because of an error during transmission. Therefore, TCP interprets timeouts as a sign of congestion and reduces the rate at which it is transmitting, it does so by setting `CongestionWindow` to half of its previous value. `CongestionWindow` is not allowed to fall below the size of a single packet, or in TCP terminology, the *maximum segment size* (MSS).
A congestion-control strategy that only decreases the window size is obviously too conservative. It also needs to be able to increase the congestion window to take advantage of newly available capacity in the network. This “additive increase” works as follows, every time the source successfully sends a couple of packets, i.e., each packet sent out during the last RTT has been ACKed, it adds the equivalent of one packet to `CongestionWindow`.

The assumptions of the congestion control mechanism in TCP, that a packet loss is due to congestion, must not necessarily be true for a wireless connection, where losses occur also due to collision, interference and other wave propagation phenomena.

Chapter 3

Integration

This chapter details the design choices for the implementation in a rather abstract way, whereas the more technical details are given in chapter 4. Extending the ANA framework to find an optimal solution for the integration of PodNet is considered in chapter 3.2. As the PodNet applications is aimed at mobile scenarios, where individuals are carrying different kinds of mobile computers, chapter 3.3 analyzes the problems inherent in platform independence.

3.1 Preliminaries

An integration of PodNet in ANA can be done in many ways. The chosen approach abides by the guidelines for the ANA framework, described in chapter 2.2, and results from the PodNet thesis [2].
The goal is to break PodNet into smaller, more manageable units, as these can be used by other applications and maintained independently of each other, essentially providing basic functional set for one-hop opportunistic applications. An advantage of the ANA framework is that it is able to adapt to different scenarios by loading and unloading functional blocks, also referred to as bricks, at runtime. By dissecting the monolithic PodNet application into a number of bricks, there may exists several versions of a brick suited for different scenarios, making the framework more robust, more flexible and more generic for the use by other applications.

A first consideration is that ANA is written in the C programming language, whereas PodNet makes use of objects in the C++ programming language. Although these languages are very close relatives, there are differences in the interpretation of code sections inherent in the two compilers for C and C++ code, which translate source code into executable programs. Therefore it may be impossible to directly include parts of the ANA library in C++ source code, and vice versa.
Another consideration is the tight bounds between PodNet and the POSIX TCP/IP socket it uses for network communication. The ANA framework does not have any similarities to a sockets workings, therefore the discrepancies between the two net-

working approaches must be apprehended and taken into account for an implementation. As a major change the version running on the ANA framework shall no longer make use of the *internet protocol* (IP) for network addressing, but instead use Ethernet addressing together with ANAs internal addressing scheme. This is because IP is intended four routing in a globally connected network, which is not needed in the opportunistic network scenario. A suitable candidate for transmission control was assumed to be found in the *store-and-forward transmission* (SAFT) protocol [35], that is already integrated in the ANA framework. But a detailed analysis revealed a very tight bound to the IP implementation for ANA, and it therefore no longer suited the demands. As an alternative to the adaptation of the SAFT protocol, implementing the *transmission control protocol* (TCP) in ANA promised to have several advantages.
The original application features both a command line version and a *graphical user interface* (GUI) for user interaction, the GUI only running on Windows Mobile devices, therefore a GUI for Linux based systems is considered. Additionally the portability to different computer architectures running Linux operating systems is heeded.

3.2 Implementation Approach

The implementation was preceded with basic C and C++ inter-operability tests, done in parallel to a familiarization with the ANA framework and the PodNet application. The message passing scheme used in PodNet, as introduced in chapter 2.3, must be redesigned as functional blocks are broken of the main application, because in ANA they do not share a common memory space for *inter process communication* (IPC), but rely on message passing methods provided by the framework. There it would be advantageous to make use of the XRP message format, which is well established in ANA, to obtain maximal compatibility.
As the PodNet application is based on older work, it contains some redundancy and unfinished ideas (e.g. router) that will be removed for simplicities sake, additionally any functionality that is already provided by the ANA framework will replace any custom made implementation in PodNet.

3.2.1 Set Of Functional Blocks

As already pointed out, the PodNet application can be divided into a *delay-tolerant networking* (DTN) part and a podcasting part. The decision on how the PodNet functions should be dissected into ANA bricks, followed the idea of maximal reuseability and flexibility. A systematically pleasing cut, adhering to the formulated ideas, splits the PodNet application into three parts: Neighbor discovery, synchronization service and the podcasting part. A graphical representation of the proposed scheme is given in figure 3.1.

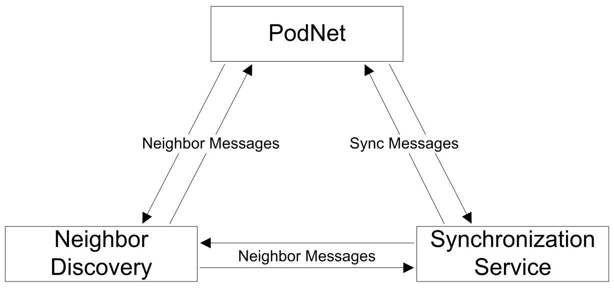


Figure 3.1: Schematic view of the planned PodNet implementation in ANA

Neighbor Discovery

The neighbor discovery module is responsible for collecting information on neighboring nodes in the network. It does so by periodically (once in two seconds) sending discovery packets to all nodes in the network, and keeping track of the packets received from other nodes.
The module informs other bricks, that are interested in the state of neighboring nodes, by sending them XRP formatted notification messages (for details see chapter 4.1.2) on any of the following events:

- **Add** Neighbor - Upon the discovery of a new neighbor node this message is sent, the new node is marked as unstable.
- **Stable** Neighbor - This message is sent, if a neighbor has been considered unstable but managed to deliver three consecutive discovery packets, and therefore gaining the stable attribute.
- **Unstable** Neighbor - A stable neighbor may become unstable, if it fails to deliver three packets in a row, this message informs about a possible link failure.
- **Remove** Neighbor - If a network node stops delivering discovery packets, it is assumed that either the link has become weak, but may regain its strength, or that the node has permanently left the network, be it because of a physical relocation or a change in it's power-state. Therefore every unstable peer is marked for removal after a given timespan (e.g. 20 seconds), once the timer expires and the neighbor is about to be deleted this message is sent.

Additionally the module maintains running totals on received packets per neighbor, allowing it to estimate a link quality by comparing these to expected values. The discovery packets can be crafted to deliver auxiliary information to the network, like for example a description of supported protocols, or other meta data.
As the discovery packets are delivered to a networks broadcast address, each participant listening to the link will read the message, but there are no assumptions made

on whether packets actually reach a neighbor or not. Such kind of reliability in message delivery is called best-effort service, and reflects the most basic characteristic of a delay-tolerant network.

Synchronization Service

So far an ANA node is up to date on its neighboring network nodes, the neighbor discovery brick ensures this by sending notification messages, on the occurrence of certain network events. As a nodes presence might be volatile, and some of its packets might have been lost, due to a temporal link failure, there must be a way of keeping track of a nodes state, i.e. whether it has changed since the last contact. The synchronization service uses two mechanisms to ensure a correct state tracking, the first being that every time the neighbor discovery brick reports a neighbor as stable, i.e. the neighbor managed to deliver three consecutive packets, the neighbor node is tagged as a potential candidate. It is of no importance if a node is new or if it regained stability after an unstable phase, the node must be inspected for its current state. The second mechanism ensuring correct state awareness, is the notification of neighboring nodes if the application that provides content, in this case the podcasting part in PodNet, reports a change in this content. This change could have been triggered by some scheduled operation or by human interaction, hence the network has no knowledge of it yet, and must be informed, either by broadcasting the information, or by individual transmission to every known node.

In order to detect changes in a neighbors state, there must be some kind of state token available for comparison, identifying individual states. The use of content timestamps as state tokens has the advantage, that it is human readable and a good indication of a neighbors actuality. The implementation follows the RFC 3339 format proposed in [2] and [36]. The exchange of tokens uses an XRP message format and must be supported by a reliable transmission protocol to guarantee proper operation. The synchronization services thus uses the the *transmission control protocol* (TCP) described in chapter 2.4 to reliably deliver synchronization messages.

PodNet On ANA

The podcasting protocol from the PodNet application, including storage management and user interaction make up the PodNet on ANA brick. Most of the functions are based on the original source code [2], actually the protocol itself remains unaltered. However, as the original relies on IP addresses for node identification, and the ANA version has its own mechanism (as explained in chapter 2.2), the protocols will be incompatible without the internet protocol available on both sides.
A registration with the synchronization service provides a set of neighbors also running PodNet on ANA, a connection to each neighbor then uses the podcasting protocol to exchange content. The podcasting protocol is also dependent on a reliable transmission mechanism, and as already mentioned, the TCP protocol

described in 2.4 serves this purpose.
Additionally the PodNet on ANA module also includes a *command line interface* (CLI) and a *graphical user interface* (GUI) for user interaction, where the CLI part is identical to the original version. These allow a user to review available channels and episodes on a network, and to manage these locally, including the issuing of new channels and episodes.

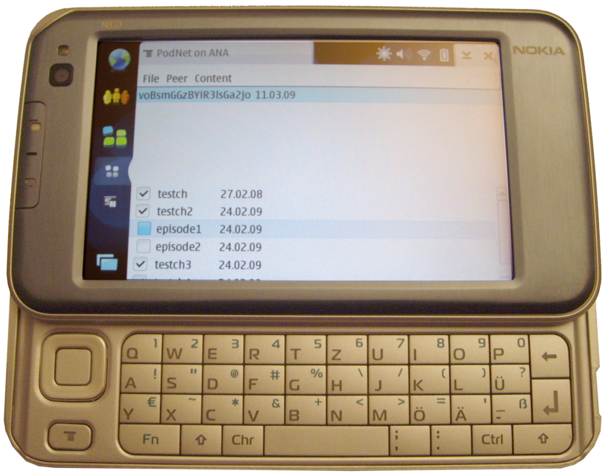


Figure 3.2: Picture of a Nokia N810 running PodNet on ANA

3.2.2 Monitoring Framework

ANA offers its own monitoring services to functional blocks. The monitoring framework aims at unifying common monitoring tasks, and to provide an adaptive and cost optimized monitoring method. The neighbor discovery brick described in chapter 3.2.1 is to be part of the group of monitoring methods related to connectivity, therefore it has to adhere to the rules for monitoring bricks, as described in detail in [39]. To give a quick overview of these basic concepts a schematic view of the monitoring framework is given in figure 3.3.
Every monitoring request to the framework is orchestrated by the dispatcher brick, it decides in witch category of monitoring tasks the current request belongs, and forwards the request to an appropriate module. This module is then responsible for a proper handling of the request, e.g. establishing a notification channel to a client brick and periodically sending notifications, and may rely on other bricks to perform the requested task. In most cases a selected monitoring category, also called monitoring metric, will not be restricted to one specific task, but will offer a variety of sub-metrics for different kinds of measurement methods and scenarios. A set of sub-metrics is depicted in figure 3.3, the neighbor discovery brick is such a sub-metric brick, it belongs to the group connectivity metrics.

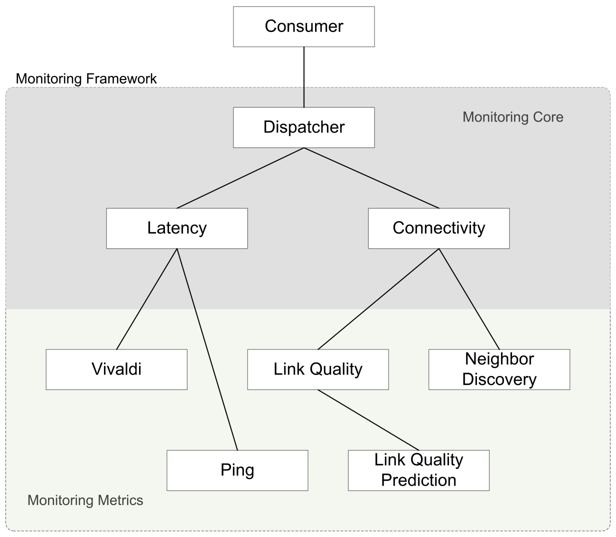


Figure 3.3: Schematic view of the Monitoring Framework in ANA

3.2.3 Addressing Scheme

As has been laid out in chapter 2.2, ANA makes use of *information dispatch points* (IDPs) to relay messages, together with a keyword published in the *key-value repository* (KVR) these two components make up an addressing scheme. The change in the usage of network protocols, away from IP packets towards Ethernet frames, has been mentioned. The ANA framework already provides an Ethernet compartment implementation, and following the requirements from chapter 2.2 the compartment implements a identifier namespace very similar to the previously mentioned KVR.

To address network nodes the employed scheme uses a pseudo unique identifier, made up of a number of random characters, in our case 20 Bytes. The use of characters reduces the possible address space, but allows human readability for a network address, furthermore the number of characters used as a network identifier must be carefully decided upon. When a network service publishes itself to the Ethernet compartment it prepends a service descriptor to the node identifier, therefore a possible network address might for example be `podnet_cLKJdsaBsdiA` or `sync_cLKJdsaBsdiA`. In the case of message broadcasting, as employed by the neighbor discovery module, there is no need for a distinguished addressing of network nodes, therefore it suffices to use a simple service descriptor, like for example `neighbors`.

3.2.4 Reliable Transmission

The availability of a reliable transmission protocol is considered as one of the key elements in the implementation of a networking application. The ANA framework

implements a transmission protocol of it's own kind, the store-and-forward transmission (SAFT) protocol. SAFT ensures a reliable transmission for hop-by-hop connections, and additionally on a end-to-end basis. The protocol relies on the *internet protocol* (IP) for node addressing and packet routing, actually the bounds are so tight, that the SAFT protocol would require major changes to become independent of IP.
It is for these reasons, that the implementation of the *tansmission control protocol* (TCP), as described in chapter 2.4, was given the preference as the reliable transmission in PodNet on ANA. The implementation should provide a byte-stream oriented protocol, similar to POSIX sockets, to simplify integration into existing projects.

3.3 Platform Independence

The network scenario for an application like PodNet is one of random mobile contacts, where people carrying mobile computers meet in the course of time, and are potentially interested in exchanging information with other available computers. Now the times, where a person carrying a mobile computer could easily be told, are over, even though wide range of today's mobile computers have only lately pervaded modern society. Today's mobile computers come in many shapes, but in general one thinks of cellular phones or *personal digital assistants* (PDAs) when speaking of popular mobile computing.
Because the energy inherent in a battery of a mobile device is very limited in comparison to a power grid's capacity, mobile computers are bound to use less capable hardware than used in *personal computers* (PCs). Often there are advantages, from an energetic perspective, when different hardware elements are united into one single chip. Also, these chips are put together with a fixed set of peripherals attached to them on a single circuit board. For it's lack of exchangeable parts and the often very specific tasks such a mobile computer is dedicated to, it is called an embedded device.
The differences in the hardware architectures of embedded devices result in the inability of using exactly the same machine code for both computers, rather each hardware platform needs a special compilation meeting its architectural parameters. From the engineering side this implies, that differences in hardware platforms must also be taken into account, when a program is designed, should the program be operable on several computers with different hardware architectures.

3.3.1 Embedded Hardware Overview

Given the number of embedded devices currently available, a mobile networking application must seek to address as many platforms as possible. Luckily for cellular phones and PDAs the number of different hardware architectures is manageable, one of most common platforms employed is the ARM [34] architecture.

The ARM architecture is an embedded 32-bit RISC CPU, most processors run around 500MHz and often support several clock speeds, to flexibly adapt performance needs versus power consumption. At its peak performance level such a processor is capable of computing 740 *million instructions per second* (MIPS). As already mentioned the CPU is hard-wired to other hardware components, for example to memory banks and other co-processing units that control access to peripheral devices. Table 3.1 shows typical examples of hardware in embedded devices, the mobile devices are the Nokia N810, a Nokia N95, the Sharp SL-C860 and Apple's IPhone.

	Sharp SL-C860	Nokia N810
CPU	ARMv5TE (400 MHz)	ARMv6 (400 MHz)
SDRAM	64 MB	128 MB
NAND	128MB	256 MB
Internal Flash	-	2 GB
External Flash	SDCard, CompactFlash	MiniSDHC
Display (Pixels)	640x480	800x480
Network	802.11b/g (CF Card)	IEEE 802.11b/g
	Nokia N95	Apple IPhone
CPU	ARMv6 (600 MHz)	ARMv6KZ (667 MHz)
SDRAM	128 MB	128 MB
NAND	147 MB	-
Internal Flash	8 GB	8GB / 16GB
External Flash	MicroSD, MicroSDHC	-
Display (Pixels)	240x320	320 x 480
Network	802.11b/g	802.11b/g

Table 3.1: Hardware specifications for the Sharp SL-C860, Nokia N810, Nokia N95 and Apple's IPhone

3.3.2 Compiling for Embedded Hardware

Often embedded devices lack the computing performance and the storage space needed for a compilation of source code. Also, they often provide a less sophisticated interface to users than operating system on the PCs. Therefore the common approach, called cross-compiling, is to make use of the knowledge about a specific target architecture, and provide a specialized cross-compilation program for PCs, which is able to generate machine executable code for that specific architecture.
A cross-compiler works just like an ordinary compiler, the only difference lies in the mapping of the set of programming instructions to the set of instructions supported by the CPU. Also the compiler may apply several architecture specific optimization steps to the code.

A variety of commercial and non-commercial operating systems is available for embedded devices, they range from time-critical, and therefore real-time capable, special purpose systems to general computing systems similar to the ones used in

personal computing. The two most widely spread general purpose operating systems for embedded devices are Windows Mobile, a reduced version of the Windows OS, and those systems that are based on the Linux operating system.
The original PodNet application was primary designed for devices running Windows Mobile, but it was also able to run on other platforms, by design. To achieve operating system independence the authors of the application needed to identify those software components that needed different interfaces on different operating systems, like for example file access or access to system timers. PodNet comes with its own platform independence library that offers an abstract interface to the system specific operations, and takes care of platform specific details, like the possibly different size of variable types in different hardware architectures.

3.3.3 The OpenEmbedded Toolchain

Modern programs are often composed of a complex set of functions that together constitute the programs overall functionality. Some of these functions are common to each computer program, and may appear in a similar form in different programs. Other functions, however, are very task specific, and depending on the programs purpose may make use of knowledge from certain fields of expertise. Therefore a set of functions, which are related by serving the purpose of solving a certain task, are combined into independent libraries. These libraries are then bound to an executable program during the compilation process, contributing the specialized functions needed by the application.
Since every library is subject to revisions, improving the functionality and eliminating flaws, the number of libraries available on a system, and the variety of possible combinations of different versions, has become confusing. A tool that pays respect to library dependencies and allows the use of different versions of the same library on the same system would drastically ease the cross-compilation process.

The OpenEmbedded toolchain [3], is a set of tools, that address these problems in cross-compilation. They take care of the issues already addressed and even allow to build complete Linux distributions, including individually selected applications besides the basic operating system. The toolchain offers numerous architectural templates, describing platform specific parameters, and some predefined distributions, that describe the operating systems software components. The wide range of supported architectures and the ability to provide distributions, as well as individual software packages for these distributions, is one of the strengths of OpenEmbedded.
As already mentioned OpenEmbedded consists of a number of tools, which are needed for cross-compilation. Strictly speaking, OpenEmbedded is composed out of two main parts, a set of rules that describe different compilation processes, and a program that interprets these rules and executes the compilation process accordingly.
This orchestrator program, called BitBake [37], is a set of scripts written in the

Python programming language. BitBake has been inspired by Portage, the software installation system for the gentoo Linux distribution. BitBake reads the platform specific configuration to instruct the cross-compiler on the target architecture, furthermore it analyzes the software packages assigned for compilation, resolves packet dependencies and includes all necessary packets in the compilation process.
A detailed description on the OpenEmbedded toolchain is given in appendix B, providing examples of different configurations.

Chapter 4

Implementation and Results

Details on the implementation of PodNet in the ANA framework are the focus of this chapter. Chapter 4.1 describes all elements that contribute to the implementation of PodNet in ANA. Chapter 4.2 presents the results of the implementation, and describes measurements in order to support the validation of the implementation.

4.1 Implementation

Chapter 3.2.1 introduced the layout of the implementations approach, and proposed a possible logical separation into separate functional blocks. Even though the concept of a functional block has already been laid out in chapter 2.2, it has only been treated theoretically, and has therefore remained somewhat obfuscated. To put things right, an example of a rudimentary ANA brick implementation is given in chapter 4.1.1, and will serve to summarize common ANA components, found in all of the brick implementations, furthermore it shows a basic network connection establishment and data sending functions.
Once the stage is set, a detailed description of implemented bricks is to follow, explaining data structures and function interfaces of every individual brick. Finnally, the resulting data flows, be it within a single functional block, or between two functional blocks on a node local level, or through a network connection, are given consideration.

As a change in comparison to previous chapters, the reference implementation of PodNet [2] will henceforth be referred to as original PodNet, whereas the implementation of PodNet on ANA will be referred to as PodNet.

4.1.1 Rudimentary ANA Example

This example makes use of the ANA API introduced in chapter 2.2.10, also described in detail in [38], and contains a description of all elements necessary to achieve basic functionality. The provided source code implements a functional block (also called a brick) for ANA, that is able to communicate with the minmex and other loaded bricks, and establishes a network connection. It is able to send and receive data in the node compartment, as well as in the Ethernet networking compartment. The source code for the example is shown in table 4.1.

```
1  #include "brick_template.h"
2
3  char *myName = "BasicBrickExample";
4  static anaLabel_t nodeIDP, netIDP, recvIDP, sendIDP;
5
6  /* Receive Node local messages and process them */
7  void recvLocal(struct anaL2_message *msg){
8    // when a messages arrives, it is of the form:
9    //     char* msg->data
10   //     int   msg->dataLen
11   ...
12 }
13 /* Receive Network compartment messages and process them */
14 void recvNetwork(struct anaL2_message *msg){
15   // when a messages arrives, it is of the form:
16   //     char* msg->data
17   //     int   msg->dataLen
18   ...
19 }
20 /* The brick_start function is the main function of each brick */
21 int brick_start() {
22   ...
23   /* we publish ourselves in the node compartement */
24   nodeIDP = anaL2_publish(NODE_LABEL, '.', myName, &recvLocal);
25
26   /* get access to the networking compartemenet (e.g. eth) */
27   netIDP = anaL2_resolve(NODE_LABEL, '.', 'eth01', 'u', myName);
28
29   /* we publish ourselves in the network compartement */
30   recvIDP = anaL2_publish(networkIDP, '*', myName, &recvNetwork);
31
32   /* get access to the network broadcast channel */
33   sendIDP = anaL2_resolve(networkIDP, '*', myName, 'b', myName);
34
35   /* send a message to the network broadcasting address */
36   char *msg = 'Hello␣World!';
37   anaL0_send(sendIDP, msg, strlen(msg));
38   ...
39 }
40 /* The brick_exit function is called before a brick is removed */
41 void brick_exit() {
42   ...
43 }
```

Table 4.1: ANA Source Code Example

A detailed inspection of the source code in table 4.1 reveals the following. On line one all necessary libraries and definitions are included, lines three and four define all the variables needed to operate in the proposed scenario. On lines six to nineteen the code defines the callback functions that will later be associated to a specific IDP, and where incoming messages will be received.

The remaining lines of code, 21 to 43, deal with the setup of variables needed for the basic operability, and by defining the two functions that every brick must implement, they are able to integrate the brick into the ANA framework. The `brick_start()` function is called upon a successfull load of a plugin by the minmex, and needs to setup a functional blocks working environment. Similarly the `brick_exit()` function needs to free the resources a brick acquired during operation, in order to be properly unloaded from the minmex.
In many cases the `brick_start()` function will be responsible for the creation of links to *information channels* (ICs) and IDPs, to provide a basic set of interfaces and associations of IDPs to callback functions. Furthermore the function may start a number of independent threads, that represent the applications main modular blocks, and is thereby able to process messages in parallel. In the case of the example presented, the code in lines 24 to 33 establishes a set of IDPs, and possibly assigns a callback function (24 and 30). Note that in line 24 the keywork `myName` gets published to the *key-value repository* (KVR) of the node compartment, and in line 30 the same keyword is published within the network compartment's namespace. Finally in the lines 36 to 37 a message is sent to all members of the network. Any node already present in the network receives this message through the `recvNetwork()` function, and may react in the way it deems appropriate.
Actually this basic example represents the core of the neighbor discovery brick described in chapter 4.1.2.

4.1.2 Neighbor Discovery

The neighbor-discovery brick's tasks have been introduced in chapter 3.2.1, additionally chapter 4.1.1 described the basic code needed for the bricks proper operability. To complete the image of the discovery process, table 4.2 shows the set of variables associated with a neighbor. The brick maintains a list of all neighbors at all times, and updates the fields related to a link's state periodically.

```
// Neighbor properties
typedef struct neighborEntry_t {
    char neighbor_id[NEIGHBOR_ID_SIZE];
    UInt32 first_seen;
    UInt32 last_seen;
    UInt32 num_packets;
    UInt32 link_quality;
    UInt32 link_bucket;

    UInt8 state;
    UInt8 old_state;
} neighborEntry;
```

Table 4.2: Neighbor Discovery Characteristics

In table 4.2 the variable `neighbor_id` identifies each neighbor unambiguously, while the variables on the remaining lines allow determining the state of a link to a neighbor, and also its quality. Additionally the last two variables are used to indicate the current state, and to check for a possible change.

As already mentioned, the neighbor discovery was to be integrated into ANA's monitoring framework, where the message format used is XRP, which itself uses a set of fields that are supported by every metric brick in the monitoring framework, these fields are shown in table 4.3.

Field	**Description**
nonce	identifies the request and is used to map results to the corresponding request
type	either a notification, a query or a subscription
type parameters	parameters like e.g. subscription age, notification thresholds, notification interval
metric	the requested metric, e.g. latency
metric parameters	predicates like e.g. requests for information on specific nodes
non-functional-parameters	attribute that influences the quality of results like e.g. tolerated error rate
replyIDP	identifies the IDP where results should be sent to

Table 4.3: ANA Monitoring Framework - Message Format for Subscriptions

The most basic subscription to the neighbor discovery metric will have to set at least the `metric` field to *connectivity*, and the `metric parameters` field to *neighbors* to enlist for neighbor notification messages, which is done by subscribing to the monitoring dispatcher. After evaluating the `metric` field of a monitoring subscription, the dispatcher decides upon which brick should the message be forwarded to, in this case it is the connectivity brick. And after the connectivity metric brick received and evaluated the message, it adds valid subscriptions to the notification list for the class of bricks related to the `metric parameters` field, and basically forwards each notification message it receives from the neighbor discovery brick to every enlisted IDP in the notification list. These notification messages are encoded in the XRP format as well, and table 4.4 shows the fields used in this message.

Field	**Description**
neighbor id	identifies a neighbor
event	a description for a neighbors current state like e.g. add, stable, etc.

Table 4.4: ANA Monitoring Framework - Message Format for Notifications

4.1.3 TCP

The implementation of the *transmission control protocol* follows the ideas introduced in chapter 2.4. One important difference to the specifications is, that all buffer pointers used to point into the receiver and the sender window are using the frame as a basic unit, whereas the specification proposes byte counts, therefore the sequence numbers also represent frame numbers and not byte numbers. This modification allows for a smaller range of sequence numbers used in a connection, simplifies computations and, by immediately scheduling any data that is written by the sending application, the scheduling algorithm as well. A consequence of this behavior is that the receiving application must, each time it reads data from the TCP socket, be willing to read at least a full frame's worth of bytes, i.e. the *maximum segment size* (MSS), as the implementation is not able to track partially retrieved frames.

The implementation splits TCP into two parts, the protocol related functions that operate on a sockets state variables on one hand, and on the other hand the part that models a socket API, operating on a set of sockets.

Protocol Core

The protocol's core can be considered as the implementation of all elements mentioned in chapter 2.4, a detailed interface listing is provided in appendix A. TCP's core element is represented by an object called `tcpState`, it contains all necessary variables to operate the protocol, its components are described in table 4.5. The functions operating on the `tcpState` object can be divided into two groups, the first encompasses all functions needed to facilitate the active state transitions in figure 2.9 from chapter 2.4, and the second group of functions that process the two message queues, i.e. the receiver buffer and the sender buffer. Upon creation of a new socket, a thread called `tcpTimedThread()` is started, that will process the send queue and react to timeouts. The `tcpProcess()` function is representing the receiver side of TCP, it is here where the extended *finite state machine* (FSM) is maintained, and where state transitions are initiated according to received packets and protocol specifications.
The TCP socket, as described in table 4.6, contains the necessary information needed to multiplex and demultiplex packets, in order to be able to maintain several concurrent connections simultaneously. In contrast to the pseudo-headers used in conjunction with TCP/IP packets, where the IP network addresses are of fixed size, the ANA implementation does not enforce a particular length on network addresses. Rather, a socket is told at initialization time, which length it shall use for network addresses in the pseudo-header. The composition of the pseudo-header and the resolution of network addresses is left to each brick that implements TCP, by design. This allows each brick to implement the kind of pseudo-header most appropriate for its networking scenario, possibly extending the header to convey other informa-

tion, and allows the use of different addressing schemes. Any implementation of the TCP protocol must therefore include the `recvLink(struct anaL2_message *msg)` and the `sendLink(tcpSocket *socket, UInt8 *data, UInt16 dataLen)` functions, containing the described multiplexing/demultiplexing processes. A pleasing aspect to the described implementation is, that TCP is fully detached from the network layer, and operates independently from the network addressing scheme.

```
//an extended tcp finite state machine
typedef struct tcpState_s {

    /* sender side state */
    tcpSeqNo     LAR;             // seqno of last ACK reveiced
    tcpSeqNo     LFS;             // last frame sent
    tcpSeqNo     LFW;             // last frame written by the application
    struct sendQ_slot {
        UInt32      timestamp;        // the moment the message was sent
        UInt8       retries;          // how many times was this segment
            transmited?
        UInt8       valid;            // msg valid?
        tcpMsg      msg;
        UInt16      msgLen;
    };
    struct sendQ_slot *sendQ;
    analock_t    sendQLock;       // mutex for the sendQ

    /* reciever side state */
    tcpSeqNo     NFE;             // seqno of next frame expected
    tcpSeqNo     LFR;             // seqno of the last frame read by the
                                  // application
    tcpSeqNo     NFR;             // seqno of newest frame received
    tcpHeader    rcvHdr;          // received header
    struct recvQ_slot {
        UInt8       valid;            // is msg valid?
        tcpMsg      msg;
        UInt16      msgLen;
    };
    struct recvQ_slot *recvQ;
    analock_t    recvQLock;       // mutex for the recvQ



    /* common */
    UInt8        FSM;             // the state of the finite state machine

    analock_t    sendWindowNotFullLock;  // the mutex for the
                                         // sendWindowNotFull semaphore
    UInt16       sendWindowNotFull;  // block the sender if the sendQ is full

    UInt16       localWindow;         // the last known local window size
    UInt8        localWindowFlag;     // did we have to drop messages?
    UInt16       remoteWindow;        // the last known remote window size
    UInt16       congestionWindow;    // the last known congestion window size
    UInt16       congestionThreshold;// the number of acks received
    UInt16       sampleRTT;           // a round trip time sample
    UInt16       estimatedRTT;        // the round trip time estimate
} tcpState;
```

Table 4.5: TCP State Variables

Another detail to the TCP implementation, and a possible difference to other implementations, is its behavior when one of the window sizes becomes zero. The original specification proposes a semaphore to control the sender window, it blocks the sending process by trapping it in a loop, should the window be full. But if, by chance, both parties of a connection run into this situation, both are blocked and there is a standoff, also called dead-lock, that requires both parties to reset their connections and start over. Therefore the described implementation restrained from using blocking functional calls, and rather informs a sending process that a queue is full, without blocking it. Of course any brick implementing TCP must consider this behavior and adapt its sending routine, by determining the status of each attempted send request.

```
//a tcp socket
typedef struct tcpSocket_t {
    /* pesudo-header information */
    UInt16      localPort;      // local TCP port
    UInt8       *localAddress;  // local network address
    UInt16      remotePort;     // remote TCP port
    UInt8       *remoteAddress; // remote network address

    /* the sockets extended finite state machine */
    tcpState    state;          // the TCP state variables

    /* control variables */
    int         closeFlag;      // indicate wheter the tcpTimedThread has to
                                // quit
    UInt32      closeTimeout;   // if in the TIME_WAIT state indicates the
                                // close timout
    UInt16      retryTotal;     // the number of retransmission timeouts
                                // occured so far
    tcpError_e  errNo;          // the error number of the last error that
                                // occured
} tcpSocket;
```

Table 4.6: TCP Socket Variables

Finally, it shall be mentioned, that the implementation supports variable sized receiver windows and sender windows, these sizes can be stated at the sockets initialization time. To support message oriented applications, like the synchronization service describer in chapter 4.1.4, the implementation supports the use of the `PUSH` flag to insert record boundaries into the byte stream. Messages with the `PUSH` flag set get delivered immediately on reception by the TCP socket, the application must implement the callback function `deliverPush()` to be able to make use of `PUSH` messages.

Socket API

To simplify the usage of the TCP service, the implementation offers a socket *application programing interface* (API), which offers all the required socket interfacing functions, and performs the tasks of maintaining the set of used sockets. The API's detailed interface listing is provided in appendix A. To illustrate the API's integration into ANA, table 4.7 provides a modification of the initial example in table 4.1. Note the changes in the name of the callback function published under recvIDP on line 34, and also the change of the channel type used in the `anaL2_resolve()` function, from 'b' like broadcast to 'u' like unicast.

In the TCP example shown in table 4.7 the first two lines establish the network addresses of the peers, assuming that these are passed to the brick as arguments, while the variable in the third line will represent the listening server. The function `deliverPush()` on lines five to nine receives network messages sent by other peers, processes them and then closes the TCP connection. The `recvLink()` function is responsible for demultiplexing packets it receives to individual sockets, furthermore it must accept new connections to valid listening ports. The message sending part in the source code is found on the lines 43 to 46, it is being preceded by the creation of a new listening server on line 40. As the program goes through these lines the `netOpen()` command associates a valid handle to the connection, used by `netSend()` to deliver the message, and finally the `netClose()` command detaches the socket from the brick. Once all frames are sent, and the remote peer has closed its side of the connection, the socket is scheduled for deletion by the API's garbage collector.

```
#include "tcpAPI.h"

char* localAddress = geAuxArg(0);
char* remoteAddress = getAuxArg(1);
tcpHandle listenServer;

//process TCP messages with the PUSH flag set
extern int deliverPush(tcpHandle handle, UInt8 *data, UInt16 dataLen) {
  ...
  netClose(handle);
}

//receive a frame from the link layer
void recvLink(struct anaL2_message *msg) {
  tcpHandle con;
  if (!(con = net_find_demux(msg->data, localAddress, remoteAddress)) &&
      net_find_socket_listen(msg->data, localAddress))
  {
    con = net_open(localPort, localAddress, remotePort, remoteAddress,
        TCP_CONNECTION_PASSIVE);
  }

  if (con) tcpProcess(con->socket, msg->data, msg->dataLen);
}

//send a frame to the link layer
extern int sendLink(tcpSocket *socket, UInt8 *data, UInt16 dataLen) {
  anaL0_send(sendIDP, data, dataLen);
}

/* The brick_start function is the main function of each brick */
int brick_start() {
  /* initialize TCP with window size=100 and networkAddressLen=4 */
  netInit(100, 100, 4, 0);
  ...
  /* we publish ourselves in the network compartement */
  recvIDP = anaL2_publish(networkIDP, '*', localAddress, &recvLink);

  /* get access to the remote network address */
  sendIDP = anaL2_resolve(networkIDP, '*', remoteAddress, 'u', myName);

  /* listen on the local port 10 for a given local address */
  listenServer = netListen(10, localAddress);

  /* send a message to the remote peer using TCP */
  char *msg = 'Hello␣World!';
  tcpHandle con = netOpen(100, localAddress, 10, remoteAddress);
  netPush(con, msg, strlen(msg));
  netClose(con);
  ...
  netFree();
}
```

Table 4.7: TCP Source Code Example

4.1.4 Synchronization Service

Like the neighbor discovery brick, the synchronization brick does also maintain a list of neighbors, based on notifications received from the monitoring framework. But unlike the monitoring brick it does not actively probe the network in any kind, it's contacts are solely based on the nodes discovered through the monitoring framework. In contrast to the neighbor discovery brick, it also uses different parameters to characterize it's peers, as shown in table 4.8.

```
// Peer properties
typedef struct syncEntry_t {
    char *neighbor_id;
    UInt32 first_seen;
    UInt32 last_seen;
    UInt32 content_timestamp;
    tcpHandle conn;
} syncEntry;
```

Table 4.8: Synchronization Service Characteristics

The synchronization service uses the addressing scheme proposed in chapter 3.2, and a similar approach to the usage of the TCP API as explained in table 4.7. Additionally to the list of neighbors, a list of potentially interested content dissemination bricks, like e.g. PodNet, is kept up to date, and every time a change in a peer's state occurs, these content dissemination bricks are sent a notification. Accordingly the synchronization service is notified of local changes in the content dissemination's data containers. The message format used, is shown in table 4.9.

Field	**Description**
neighbor id	identifies a neighbor
timestamp	the neighbors content timestamp

Table 4.9: Message Format for Synchronizations

4.1.5 PodNet

The PodNet implementation is a hybrid, the networking functionality provided by ANA being a C libray, and the podcasting protocol and the main PodNet application being implemented in C++. This setup requires some mutual interface functions, as the ANA API can not be included in the C++ code, due to the strictness of the compiler. A detailed listing of these interface functions and the modular setup is described in appendix A.
To give an overview of the overall workings of all the bricks implemented so far, and to illustrate their relation to each other, figure 4.1 shows a schematic view of the implementation. It is worth mentioning that the schematic view simplifies the

interactions with the ethernet compartment, specifically the different *information dispatch points* (IDPs) established for inter-brick communication with the ethernet brick. A comparison with the ethernet compartment, shown in more detail in figure 2.5, reveals the full picture.

The PodNet brick encompasses the user interfaces, i.e., the *command line* tool and the *graphical user interface*, both are directly controlled by the brick. A detailed description on the PodNet modules and protocol specifications is given in [2].

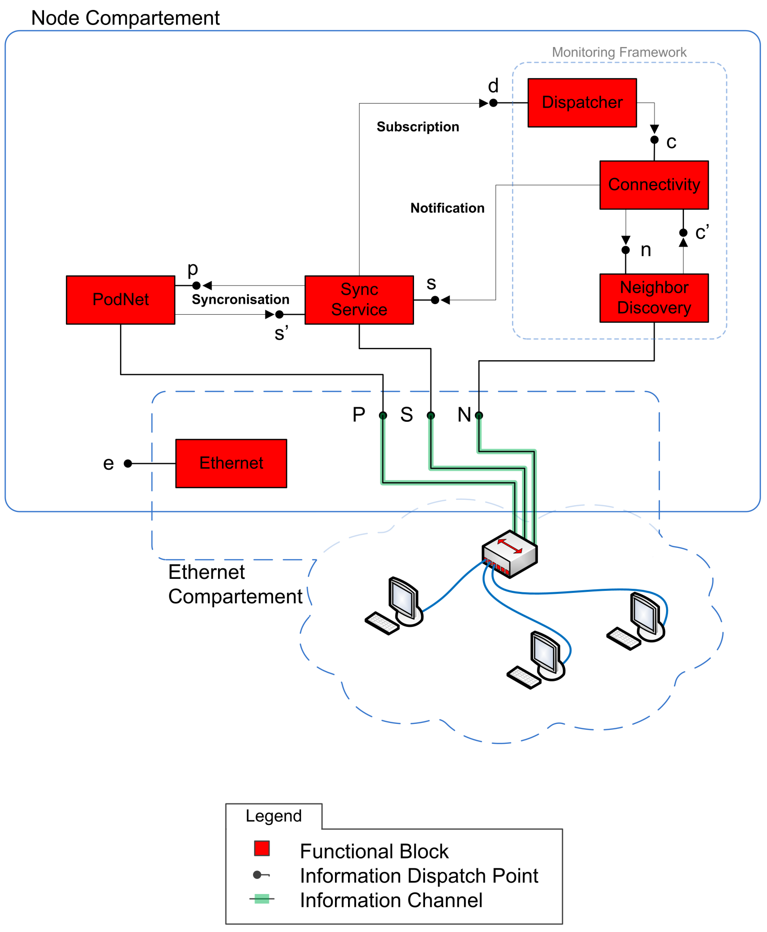


Figure 4.1: Schematic View of the PodNet on ANA Implementation

4.2 Results

In order to evaluate the implementation, and to get an idea about the performance of PodNet on ANA, and ANA itself, different measurements shall be discussed in this chapter. All of the following diagrams were generated by recording the TCP state variables associated with a socket. Besides the pointers into the circular buffers, i.e. the sender and the receiver window, other variables of interest, as for example the congestion level, or the estimated *round trip time* are followed as well.

4.2.1 Methodology

The measurements were conducted on two IBM Thinkpads T42 (1.8 GHz PentiumM), in conjunction with a Nokia N810 (see table 3.1).
To inspect the TCP implementation, a first experiment transfers bulks of random data between two IBM Thinkpads personal computers. There are three different kinds of network connections that have been considered, these interfaces are:

- Loopback Interface: The virtual network device present in all TCP/IP operating system implementations, which is fully integrated into the computer system's internal network stack. In most cases it emulates the network connection by piping signals through local *first-in-first-out* (FIFO) buffers, on the link layer.

- LAN (IEEE 802.3): Ethernet is the most widely used network technology on wired networks (LANs) today, therefore the TCP protocol should be tested in this setup, a twisted-pair cable was used to establish the connection between the IBM Thinkpads. If the network connection passes through a network switch, in contrast to a network hub, then a fast wired link is very much comparable to the loopback interface, in expected packet losses and performance values.

- WLAN (IEEE 802.11b): This wireless network standard allows extending wired Ethernet networks, by connecting a wireless *access point* (AP) to the LAN, and thereby provides location freedom to network nodes. As an alternative, where no fixed infrastructure is available, network nodes may meet in the *ad-hoc* network mode, and use ZeroConfiguration [40] to setup a network. The wireless medium, as the wired scenario where nodes are connected by a hub, is shared medium, i.e. data that is put on the link is perceived by every participant in the network. But unlike wired scenarios, WLANs additionally suffer from packet loss due to radio interference, and other wave propagation phenomena.

The TCP test application makes use of the neighbor discovery brick to find its peer, once found it launches a connection and a associated timer, then it periodically tries to write a certain number of packets into the send buffer. The thread responsible for processing the TCP send buffer then produces measurements, by writing the state variables representing the TCP circular buffers, as shown in figure 2.8, on each periodic execution of the thread into a file using *comma-separated values* (CSVs). The following state variables get tracked:

- Time: Recorded Timestamp
- LAR: Sequence number of the Last Acknowledgement (ACK) Received
- LFS: Sequence number of the Last Frame Sent
- NFE: Sequence number of the Next Frame Expected
- LFR: Sequence number of the Last Frame Read by the application
- Remote Window: Size of the Remote peer's receiver Window
- Congestion Window: Size of the Congestion Window
- Retransmissions: Number of total Retransmission occurred so far

The diagrams shown in chapter 4.2.2 were obtained by evaluating the CSV files, which were generated for each networking interface. The TCP connections used a sender buffer of 100 packets, which is equal to $100 \times MSS$ (maximum segment size) bytes, and a receiver buffer sized 1000 packets, worth $100 \times MSS$ bytes.

The second measurement performed is aimed at a comparison of the ANA version of PodNet with the original implementation in [2], where Wacha et al. had a testbed of 25 Compaq IPaqs at their disposal. The goal is to compare the results of one single connection, and then by induction, make an estimate on a groups performance. In analogy to the tests conducted in [2], the first scenario transfers ten channels, with two to three episodes each. There are no file enclosures attached to these episode. The file transmission capability is examined in the second scenario. These experiments were run using the IBM Thinkpad and the Nokia N810.

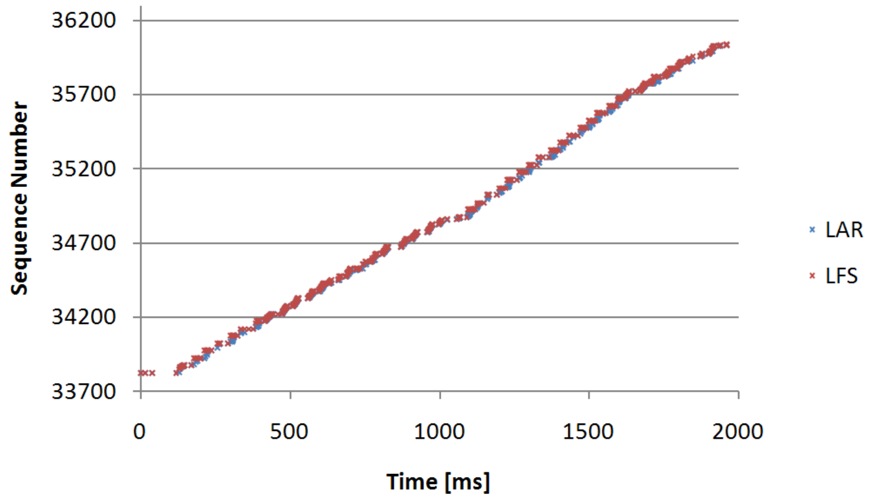
4.2.2 Measurements, Results and Discussion

The data produced by the experiments, as explained in chapter 4.2.1, was refined into significant diagrams. These diagrams are presented and discussed in the following.

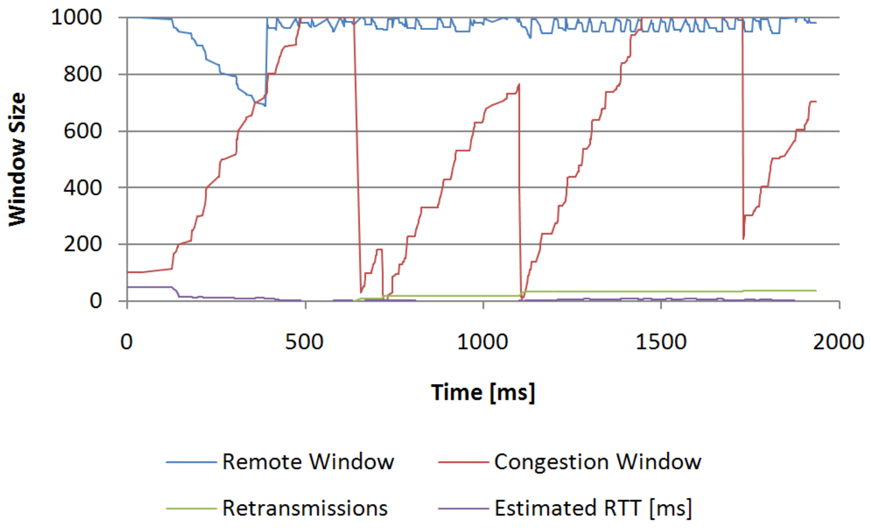
4.2.3 TCP Measurements

The first diagrams, numbered one, in figure 4.2 and figure 4.3 show the development of the sender side variables *Last ACK Received* (LAR) and *Last Frame Sent* (LFS). Both plot sequence numbers against time, but use different network interfaces to connect the peers, which explains the different slopes.
The second diagrams, numbered two in figures 4.2 and 4.3, show the window sizes

of the remote window and the congestion window, maintained in each TCP connection, and the number of retransmission that occurred during the elapsed interval. Additionally the estimated RTT is shown to help visualizing the timing in the transmission process.



1.



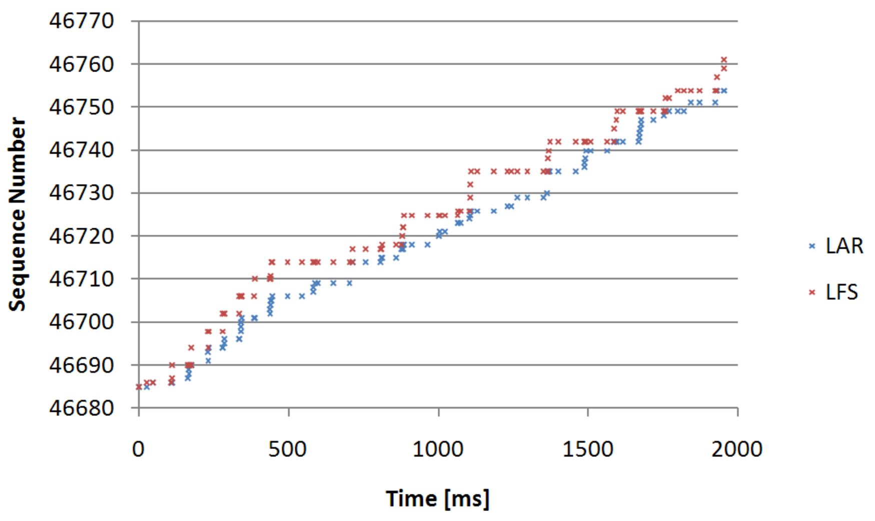
2.

Figure 4.2: TCP Cable-Measurement Results; 1. Cable Sequence Numbers, 2. Cable Window Sizes

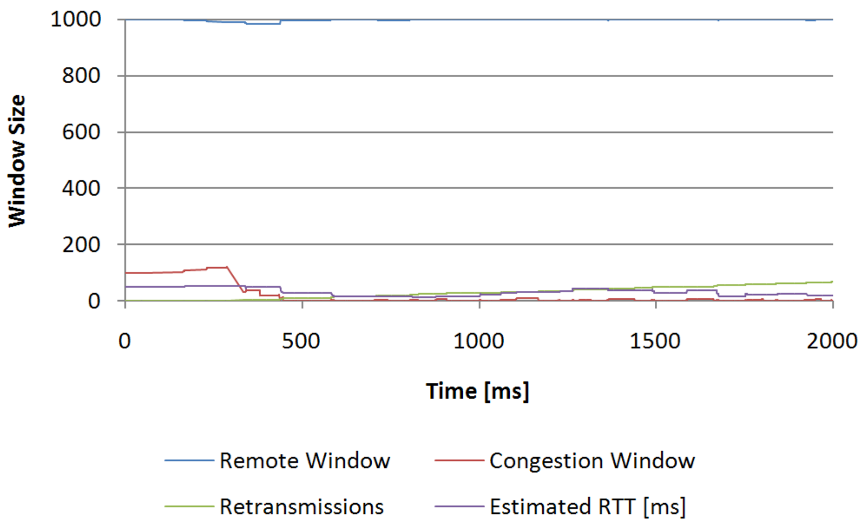
Figure 4.2 shows the circular buffer results for the wired network connection. The

evolution of the LAR and LFS variables are much more in phase than those in figure 4.3. This is the direct consequence of the wired networks theoretical property, that no packets are lost in transmission. Furthermore the slope of the graph is representative for the transmission rate the TCP protocol achieves, the steeper the rise, the faster the packets get delivered to their destination.
An interesting observation to the wired scenario in figure 4.2 is, that altough the interface is lossless, a few packet retransmissions do occur during the transmission. This is most likely attributed to the fact, that the minmex process runs in the userland of the operating system, and is therefore not capable of processing data in a near real-time manner, as opposed to kernel functions. It is the operating system that distributes the CPU's resources at its discretion, and according to the current processor load. It is therefore possible, that although a network frame arrived in perfect order, the ANA framework is not able to process it, and therefore a timeout occurs.
The sequence of circular-buffer variables in 4.3 reveals the links susceptibility to congestion. The difference between the LAR and LFS variables, the LFS variable is always leading, illustrates the number of packets the TCP socket has currently under way, or put another way, the usage of the sender window. The step pattern of the LFS variable is induced by TCP congestion control mechanism, that regulates the sending rate. In the beginning of the connection (before 400 ms) the link is not yet congested, therefore the variables lie closer to each other. Then as the load on the link increases, and packet losses occur, TCP has to wait until it receives enough AKSs to reopen the congestion window.
In general, figure 4.2 presents the results of an optimal loaded link, whereas figure 4.3 shows the results of a congested or overloaded link.

The inner workings of the congestion control mechanism are best understood when looking at the evolution of the $congestionWindow$ variable in figure 4.2. The initial value of 100 is slowly increased as ACKs arrive at the sender until the maximum value of 1000, then at approximately 600 ms and 1100 ms packet losses occur, causing the window to shrink all the way down to zero, effectively stalling the sender. This is also visible in the circular-buffer plot in figure 4.2, where the slope of the curves slightly decreases its steepness. Additionally the number of $retransmissions$ is increased each time a packet loss occurs.
The variables showing the $estimatedRTT$ and the $remoteWindow$ are providing an informative value of the current network load (estimated RTT tends to increase on loaded links) and the state of the remote peer, i.e., it's rate of data retrieval.



1.



2.

Figure 4.3: TCP WiFi-Measurement Results; 1. WiFi Sequence Numbers, 2. WiFi Window Sizes

To illustrate the performance of the TCP implementation in the ANA framework, several measurements with different data rates were conducted on a certain link. The data throughput was recorded during the transmission, allowing to put the load into relation with the throughput.

The results of these tests are summarized in figure 4.4. The axes show values relative to the links theoretical maximum achievable throughput. For the 100Mbit/s LAN connection, it is 12.5 MB/s, and for the wireless 11Mbit/s connection, this amounts to 1.3 MB/s. The results show that for both links there is an increasing discrepancy between the ideal line and actually measured throughput, as the load increases.

As already mentioned, a minor deviation from the ideal line is explainable by the way ANA is executed in the operating system, as a user process with no real-time capabilities.
The massive drop in the performance of the wireless link, however, is not related to this issue. The drop is partially due to the lossy nature of the link, but can not solely be attributed to it, and therefore a detailed investigation of TCP's behavior in wireless networks should be considered as a next step for future work. We think that it has something to do with packet timing, more precisely with the way the TCP implementation puts packets on the link. Currently several packets are sent in a row through the ANA framework, with no delays in between, therefore bursting the link if we assume that there is no further delay introduced by the operating system. Therefore a possible solution could be, to allow sending only a few packets at once, and in the meanwhile, reducing the time interval for packet processing.

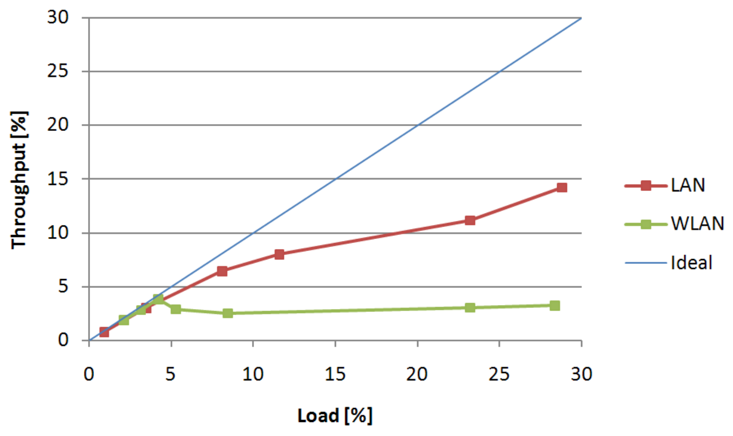
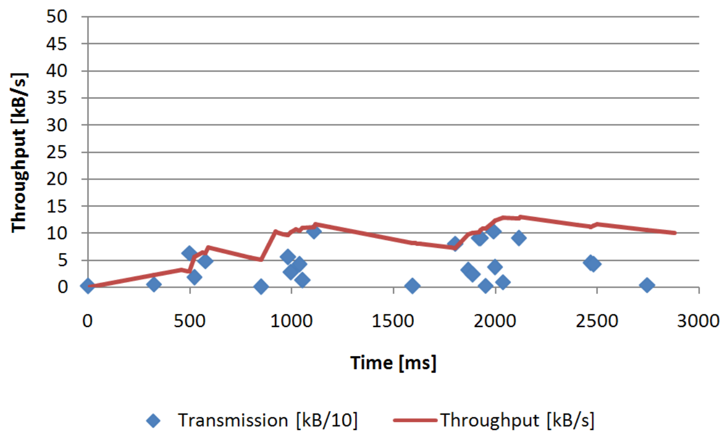


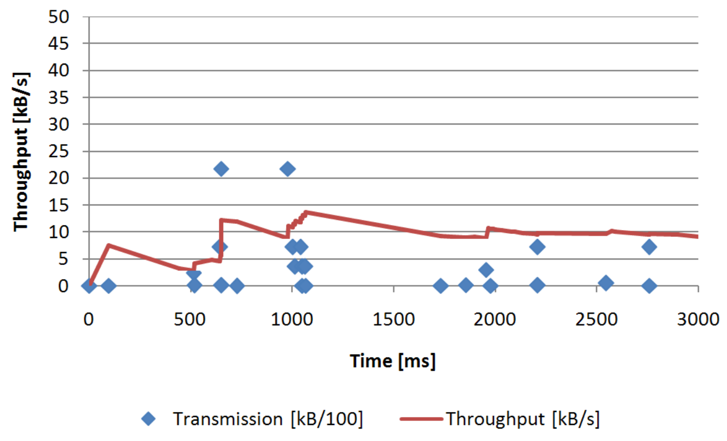
Figure 4.4: TCP Load/Throughput Diagram

4.2.4 PodNet on ANA Tests

To evaluate the PodNet implementation, and to put it into relation with the original application in [2], several tests were performed. The measurement serve to illustrate the implementations performance.



1.



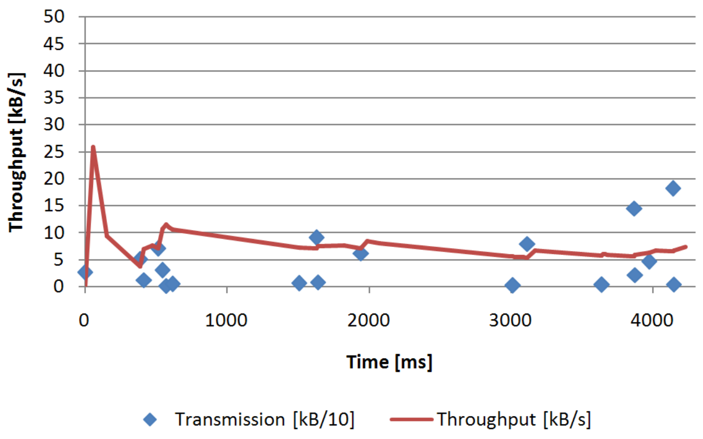
2.

Figure 4.5: PodNet Measurement Results - Episodes Synchronization - Sending Process; 1. Nokia N810, 2. Thinkpad T42

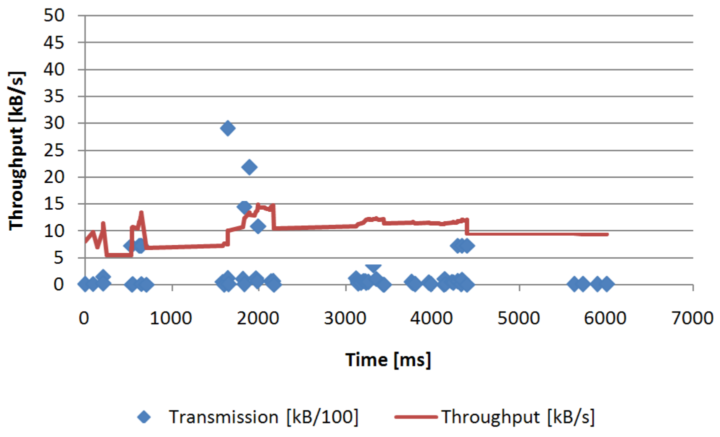
Both tests use the wireless network interface as the physical link, the first device is the Nokia N810 and the second a Thinkpad T42.

The first test synchronizes two devices, the data that is exchanged is solely composed of channel and episode data (10 channels, each having 3 episodes), with no additional file attachments. The results, given in figure 4.5, show the throughput of each connection and the individual packet transmissions that contribute to the average throughput.

The second test does not only synchronize two devices, but also transferres a file enclosure of an episode (50 kBytes). The results of this process are shown in figure 4.6. Again individual transmissions form the shape of the average throughput.



1.



2.

Figure 4.6: PodNet Measurement Results - File Transfer - Sending Process; 1. Nokia N810, 2. Thinkpad T42

Both test results motivate the assumption, that the bursty nature of the TCP implementation is causing the poor link utilization. One can see, that the throughput first rises very quickly, then congestion occurs, which causes the performance drop. TCP counters this behavior by increasing the *packet timeout*, and thus performs as expected. The most promising point, however, is that the effect is more pronounced for the Thinkpad T42 than for the Nokia N810, which indicates, that it is indeed related to how quick packets are put to the physical link, as the Thinkpad operates at a much higher speed than the Nokia N810.

Chapter 5

Conclusion

This chapter summarizes the contributions provided by this thesis in 5.1, and relates them to the initial problem setting in chapter 1. The final section 5.2 analyzes current shortcomings of the implementation, and gives an outlook to possible future extensions.

5.1 Contributions

The implementation of the DTN facilities, found in the 'neighbor discovery' and the 'synchronization service' brick and described in detail in chapters 3 and 4, enables ANA [9] to be aware and make use of opportunistic networking contacts and the state of synchronicity of data for these contacts. Furthermore, by integrating these new bricks into the monitoring framework of ANA, they can be used in different networking scenarios, like a wired or wireless ethernet network (but also for example Bluetooth), out of the box. It is even possible to seamlessly switch from one technology to another at runtime, therefore enabling a mobile device to choose the most promising out of the available alternatives. The evaluation of this thesis showed, as described in chapter 4.2, the resilience of the implementation, even under heavy network load. The scheme, proved as well designed, can therefore serve as a platform for further opportunistic functionality in ANA.
The proof-of-concept implementation of PodNet in ANA makes use of the TCP protocol implementation. Therefore it is also resilient to packet losses, and by providing POSIX socket-like interfaces for TCP, it relieves application developers of taking care of the transmission process itself. The PodNet on ANA implementation enhances the original PodNet application [2] with a *graphical user interface* for Linux systems, and the ability to run on handheld Linux devices. The implementation was successfully tested on the Nokia N810 and the Sharp SL-C860, proving platform independence for the ANA framework.

5.2 Future Work

As already described, the implementation proved its resilience in different networking scenarios (LAN, WLAN), i.e., it is able to cope with suboptimal network links, and guarantees reliable delivery. The evaluation tests showed, however, that the applications performance on a wireless link is not yet optimal (as shown in figure 4.4 in chapter 4.2). The reason for this discrepancy is not very well understood, and motivates future research in this topic.
The portability of the ANA framework on to handheld devices has been explored with positive results, but the range of tested devices was rather small. Therefore it would be interesting to explore different, and foremost more recent, devices like e.g. the Apple IPhone, or Google's Android mobile phone.

Additionally to the possibilities of exploring different mobile devices, and the improvements of the frameworks foundation, there are numerous possibilities for a possible functional expansion:

- Extensions to the framework
  - User Notion: The notion of a user would allow to introduce content rating schemes, and to relate network contacts to a person.
  - Neighbor Groups: In the case of a high density in network neighbors, a logical separation into different user groups could provide a way to reflect different trust levels between individual users.
  - Content groups: Different content merits different treatment, for example a user could be interested in music only, and therefore a logical separation of content would allow to filter for different types of informations.
- Applications for the framework
  - Partner Search: Look for possible partners in the vicinity of the user, and raise an alarm if a perfect match has been detected, possibly also lowering the current divorce rate.
  - Flea Market: Place advertisements for items that are no longer in use, and find items of interest in the users vicinity, that could be worth inspecting.
  - File Sharing: Extend the classical file sharing applications to allow users to exchange data of interest, as e.g. music files.
  - Friendship Communities: As mentioned in the introduction, the online communities could be enhanced, to allow the detection of friends in the vicinity.
  - Micro Blog: Additionally to friendship communities, these could be extended with micro blogs, where users can state in a short sentence, whats on their mind.

The applications proposed for future work in the field of opportunistic networking all have in common, that they propose classical Internet concepts, for the mobile scenario. In the current stage, such mobile applications are still in the minority, and it is not quite clear in which direction they will evolve.
It can be taken for granted, however, that mobile applications will be closely interweaved with modern society, reflecting and influencing the social behavior of the present time.

Although modernization processes can not be halted, it is important to maintain a critical perspective to these events also. Nowadays it is no challenge to locate the position of a mobile phone, and to remotely inspect the data stored in it, independent of its location. The more data these devices hold in store, be it personal or business related, the more information about the individual carrying this data can be obtained.
Besides all the positive influences, the exposure of critical data may also lead to dire consequences for the individual. Society will have to find an optimal balance between these two extremes, a challenging task for the near future.

Bibliography

[1] The aka aki network. Online: http://www.aka-aki.com/ and http://www.iht.com/articles/2008/09/12/business/aka.php

[2] C. Wacha. Wireless Ad Hoc Podcasting with Handhelds. Master Thesis MA-2007-05, TIK Communication System Group, ETH Zurich, April 2007.

[3] The OpenEmbedded project. Online: http://www.openembedded.org/

[4] E. Kohler, R. Morris, B. Chen, J. Jannotti and M.F. Kaashoek. The Click Modular Router. ACM Transactions on Computer Systems (TOCS), Volume 18, Issue 3, August 2000, pp. 263-297.

[5] V. Lawadia, Y. Zhang, B. Gupta. System Services for Ad-Hoc Routing: Architecture, Implementation and Experiences. In Proc. of MobiSys 2003: The First International Conference on Mobile Systems, Applications, and Services, San Francisco, USA, May 2003, pp. 99-112.

[6] Delay Tolerant Network Research Group. Online: http://www.dtnrg.org/

[7] Seven Degrees of Separation. Columbia University, NewYork, USA, 2002. Online: http://www1.cs.columbia.edu/ arezu/7DS/ and http://www.cs.unc.edu/ maria/7ds/

[8] The Haggle project. Online: http://www.haggleproject.org/

[9] C. Tschudin et al. ANA Blueprint. Online: http://www.ana-project.org

[10] The TCP specifications. Online: http://tools.ietf.org/html/rfc675

[11] A. Balasubramanian, B.N. Levine, and Arun Venkataramani. DTN routing as a resource allocation problem. In Proc. ACM SIGCOMM Applications, Technologies, Architectures, and Protocols for Computer Communication, Kyoto, Japan, August 2007, pp. 373-384.

[12] S. Jain, K. Fall, R. Patra. Routing in a Delay Tolerant Network. ACM SIGCOMM Computer Communication Review, Volume 34, Issue 4, 2004, pp. 145-158.

[13] D. Jea, A. Somasundara, and M.B. Srivastava. Multiple Controlled Mobile Elements (Data Mules) for Data Collection in Sensor Networks. In Proc. of IEEE/ACM International Conference on Distributed Computing in Sensor Systems (DCOSS), June 2005.

[14] S.C. Rahul, R. Sumit, J. Sushant, and B. Waylon. Data MULEs: Modeling a Three-tier Architecture for Sparse Sensor Networks. In Proc. of IEEE SNPA Workshop, May 2003.

[15] Avriel, Mordecai (2003). Nonlinear Programming: Analysis and Methods. Dover Publishing. ISBN 0-486-43227-0

[16] V.D. Park, M.S. Corson. A highly adaptive distributed routing algorithm for mobile wirelessnetworks. In proc. of INFOCOM 97 - Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies, Kobe, Japan, April 1997, pp. 1405-1413.

[17] M. Ammar, M.M. Bin Tariq, E. Zegura. Message ferry route design for sparse ad hoc networks with mobile nodes. In Proc. of the 7^{th} ACM international symposium on Mobile ad hoc networking and computing, Florence, Italy, 2006, pp. 37-48.

[18] A. El Fawal, J. Le Boudec, K. Salamatian. Self-Limiting Epidemic Forwarding. Technical Report LCA-REPORT-2006-126, EPFL, Lausanne, Switzerland, 2006.

[19] T. Spyropoulos, K. Psounis, and C.S. Raghavendra. Spray and wait: An efficient routing scheme for intermittently connected mobile networks. In WDTN 05: Proceeding of the 2005 ACM SIGCOMM workshop on Delay-tolerant networking, Philadeplhia, USA, 2005, pp. 252-259.

[20] A. Lindgren, A. Doria, and O. Scheln. Probabilistic routing in intermittently connected networks. In the SIGMOBILE Mobile Computing and Communications Review, Volume 7, Number 3, July 2003, pp 19-20.

[21] C. Becker and G. Schiele. New Mechanisms for Routing in Ad Hoc Networks. In proc. of 4^{th} Plenary Cabernet Workshop, Pisa, Italy, October 2001.

[22] E.M. Daly, M. Haahr. Social network analysis for routing in disconnected delay-tolerant MANETs. In Proc. of ACM SIGMOBILE International Symposium on Mobile Ad Hoc Networking & Computing, Montreal, Canada, 2007, pp. 32 - 40.

[23] Milgram. Small World Experiment. Online: http://www.stanleymilgram.com/milgram.php

[24] A.L. Barabasi. Linked: How Everything Is Connected to Everything Else and What It Means for Business, Science, and Everyday Life. Penguin Group USA, New York NY, April 2003.

[25] J.D. Watts. Small Worlds: The Dynamics of Networks Between Order and Randomness. Princeton University Press, 1999.

[26] L.A.N. Amaral, A. Scala,M. Barthélémy, and H. E. Stanley. Classes of small-world networks. In proc. of the National Academy of Sciences, 2000.

[27] ANA Project - Autonomic Network Architecture. Online: http://www.ana-project.org/

[28] D. Clark, J. Wroclawski, K. R. Sollins and R. Braden. Tussle in Cyber-space: Defininf Tomorrow's Internet. In Proc. of ACM SIGCOMM, Pittsburg, PA, USA, August 19-23, 2002, pp. 347-356.

[29] D. Clark, R. Braden, A. Falk and V. Pingali. FARA: Reorganizing the Addressing Architecture. In Proc. of ACM SIGCOMM Workshop on Future Directions in Network Architecture (FDNA), Karlsruhe, Germany, August 2003, pp. 313-321.

[30] J. Pujol, S. Schmid, L. Eggert, M. Brunner and J. Quittek. Scalability Analysis of the TurfNet Naming and Routing Architecture. In Proc. of ACM 1st ACM Workshop on Dynamic Interconnection of Networks (DIN 2005), Cologne, Germany, September 2, 2005, pp. 28-32.

[31] B.H. Bloom. Space/time trade-offs in hash coding with allowable errors. Communications of the ACM 13 (7), 1970, pp. 422?426.

[32] The OBEX protocol specifications. Online: http://www.irda.org/

[33] L.L. Peterson, B.S. Davie. Computer Networks - A Systems Approach. Morgan Kaufmann Publishers, San Francisco, United States of America, 2003, pp. 374 - 405.

[34] The ARM hardware architecture. Online: http://www.arm.com/

[35] The SAFT protocol specifications.
Online: http://people.ee.ethz.ch/˜simonh/research/saft/start

[36] The RFC3339 specification. Online: http://tools.ietf.org/html/rfc3339

[37] The BitBake software. Online: http://developer.berlios.de/projects/bitbake

[38] G. Bouabene, C. Jelger, A. Keller. ANA Core. Online: http://www.ana-project.org

[39] C. Tschudin et al. ANA Monitoring. Online: http://www.ana-project.org

[40] ZereConf for MANETs. Online: http://de.wikipedia.org/wiki/Zeroconf

Appendix A

Implementation Details

A.1 Overview

The listing of all files related to the ANA framework serves as an index to this appendix. All relevant files are highlighted and will be shortly discussed in the corresponding section.

Furthermore these dependencies must be satisfied, in order to be able to develop the software:

- subversion
- kdevelop (the authors IDE)
- build-essential (e.g. from ubuntu)
- libncurses5-dev
- libgtk2.0-dev

```
ana-core/devel/

- ana_podnet_init.sh - A.2.1
- config.txt - A.2.1
+ C
  + bricks
    + API
    + dtn
      - Makefile-user
      + podnet
        - bloom_filter.h
        - bloom_filter.cpp
        - config.h - A.2.2
        - datastore.h
        - datastore.cpp
        - debug.h
        - flexpacket.h
        - flexpacket.cpp
        - Makefile
        - message.h - A.2.2
        - message.cpp
        - podnet.c - A.2.2
        - podnetGui.h - A.2.2
        - podnetGui.cpp
        - podnetMain.h - A.2.2
        - podnetMain.cpp
        - rc_commands.h
        - rc_commands.cpp
        - transfer.h - A.2.2
        - transfer.cpp
        - xrpMessage.h - A.2.2
        - xrpMessage.cpp
        + tinyxml
          - ... XML Library ...
        + PI
          - ... Platform Indepen-
                dence Library ...
      + shared
          - Makefile
          - tcpAPI.h - A.2.3
          - tcpAPI.c
          - tcp.h - A.2.3
          - tcp.c
          - testtcp.c - A.2.3
      + syncdiscovery
          - Makefile
          - syncdiscovery.c - A.2.4
          - syncManager.h - A.2.4
          - syncManager.c
    - mcis
    + monitoring
      - Makefile-user
      + connectivity
        - connectivity.c
        + neighbors
          - config.h - A.2.5
          - neighbordiscovery.c - A.2.5
          - neighborManager.h - A.2.5
          - neighborManager.c
      + dispatcher
        - dispatcher.c

+ C
  + bricks
    - ...
    - vivaldi
    - cfinder
    - eth-vl
    - vlink
    - chat
    - ip
    - saft
    - tools
  + include
    - anaCommon.h
    - anaError.h
    - anaLib0.h
    - anaLib1.h
    - anaLib2.h
    - analock.h
    - anaThread.h
    - anatimer.h
    - anaValidate.h
    - ana_vlinkAPI.h
    - brick_template.h
    - listAPI.h
    - minmex_decl.h
    - quickRepository.h
    - xrp.h
  + minmex
    - ... Minmex
          implementation ...
  + shared
    - anaCommon.c
    - anatimer.c
    - xrp.c
    - anaThread.c
+ bin
  - minmex - A.2.6
  - mxconfig - A.2.6
  - vlconfig - A.2.6
+ so
  - agnostic_chat.so
  - cfinder.so
  - connectivity.so - A.2.7
  - dispatcher.so - A.2.7
  - eth-vl.so
  - neighbordiscovery.so - A.2.7
  - podnet.so - A.2.7
  - syncdiscovery.so - A.2.7
  - testtcp.so - A.2.7
  - vlink.so
```

A.2 Descriptions

A.2.1 ANA Environment

Files that are not mentioned in this chapter, are either provided by the ANA framework itself, or were copied one-to-one from the original PodNet application [2].

ana_podnet_init.sh

A shell script that loads all the required bricks into the minmex, and binds the desired network interface. The script requires 4 arguments, therefore the usage is

```
ana_podnet_init network-interface[eth0/wlan0] GUI[0/1] instances runsingle
```

The `instances` argument, defines how many local instances we want to start, and the variable `runsingle` can either be set to `all` to start every instance or a number less or equal to `instances` to start a specific instance.

config.txt

Configuration file for the ANA framework, describes which bricks should be compiled

A.2.2 PodNet

config.h

The PodNet configuration file, describes constants as directories and other running parameters.

message.h

The original file was modified, local messages now make use of the XRP file format. Network messages were adapted, IP addressing was replaced by ANA addressing, as described in chapter 3.2.3.

podnet.c

The ANA framework brick-implementation, establishes all *information dispatch points* (IDPs) relevant for communication, and takes care of the pseudo-header multiplexing / demultiplexing process.

podnetGUI.h

The Gtk2 GUI for the Linux operating system. It is tightly bound to the rest of the application.

podnetMain.h

The glue in the PodNet on ANA implementation, connects the original modules from [2] with the parts that were newly designed during this thesis.

transfer.h

The original file was modified, IP addressing was replaced by ANA addressing, as described in chapter 3.2.3.

xrpMessage.h

The implementation of the XRP message format offers object-oriented C++ interfaces, otherwise identical to the C version from `ana-core/devel/C/include/xrp.h`.

A.2.3 TCP

tcpAPI.h

The TCP *application programming interface* (API) offers following interface functions:

```
//--------------------
// Core API functions
//--------------------
tcpHandle netOpen(UInt16 localPort, UInt8 *localAddress, UInt16 remotePort,
    UInt8 *remoteAddress);
tcpHandle netListen(UInt16 localPort, UInt8 *localAddress);
tcpHandle netAccept(tcpHandle handle);
int netClose(tcpHandle handle);

int netReceive(tcpHandle handle, UInt8 *data, UInt32 dataLen);
int netSend(tcpHandle handle, UInt8 *data, UInt32 dataLen);
int netPush(tcpHandle handle, UInt8 *data, UInt32 dataLen);

int netIsOpen(tcpHandle handle);
int netIsClosed(tcpHandle handle);
int netIsError(tcpHandle handle);

//-----------------------
// Environment functions
//-----------------------
int netInit(UInt16 recvWindowSize, UInt16 sendWindowSize, UInt8
    networkAddressLen, UInt8 options);
int netFree();

//------------------
// Helper functions
//------------------
int net_header_ports(tcpMsg msg, UInt16 *srcPort, UInt16 *dstPort);
tcpHandle net_open(UInt16 localPort, UInt8 *localAddress, UInt16 remotePort,
    UInt8 *remoteAddress, UInt8 options);
tcpConnection* net_find_demux(tcpMsg msg, UInt8 *localAddress, UInt8 *
    remoteAddress);
tcpConnection* net_find_socket_listen(tcpMsg msg, UInt8* localAddress);
tcpConnection* net_find_socket(tcpSocket *socket);
tcpConnection* net_find_handle(tcpHandle handle);
```

```
32
33 //--------------------------------------
34 // EXTERNAL CALLBACK FUNCTIONS
35 // -------------------------------------
36 //callback function for transmission control when sending a frame to the link
       layer
37 int sendLink(tcpSocket *sock, UInt8 *data, UInt16 dataLen);
38 //callback function for transmission control when forwarding a PUSH message
       to the higher-level-protocol
39 int deliverPush(tcpHandle handle, UInt8 *data, UInt16 dataLen);
```

The interface functions on lines 1-21 are self-explanatory, the helper functions on lines 23-31 are used in conjunction with multiplexing / demultiplexing process, located in the brick implementation file. The brick implementation also needs to implement 2 mandatory functions, i.e. the `sendLink()` that puts data on the link (including multiplexing) and the `recvLink()` function, demultiplexing the TCP stream and forwarding it to the TCP packet processor.
Additionally the brick may choose to implement the `deliverPush()` function, if it wants to make use of record boundaries, as described in detail in chapter 2.4.4.

tcp.h

The core of the TCP protocol implementation, may also be used without the `tcpAPI.h`, if the developer wishes so. The core provides following functional interface:

```
//-------------------------
//TCP Core functions
//-------------------------

// initialization
// ---------------------
void tcpConfig(UInt16 recvWindowSize, UInt16 sendWindowSize, UInt8 options,
     UInt8 networkAddressLen);
void tcpInit(tcpSocket *socket);

// process outgoing packets on a TCP socket
// ---------------------------------------------
static void tcpTimedThread(tcpSocket *socket)

// process incoming packets on a TCP socket
// ---------------------------------------------
int tcpProcess(tcpSocket *socket, tcpMsg msg, UInt16 msgLen);

// open, close and operate tcp sockets
// -------------------------------------
int tcpOpen(tcpSocket *socket, UInt16 localPort, UInt8 *localAddress, UInt16
     remotePort, UInt8 *remoteAddress);
int tcpListen(tcpSocket *socket, UInt16 localPort, UInt8 *localAddress,
     UInt16 remotePort, UInt8 *remoteAddress);
int tcpReceive(tcpSocket *socket, UInt8 *data, UInt32 maxDataLen);
int tcpSend(tcpSocket *socket, tcpFlag_e flags, UInt8 *data, UInt32 dataLen);
int tcpClose(tcpSocket *socket);
int tcpReset(tcpSocket *socket);

// evaluation functions for tcp sockets
// -------------------------------------
extern int tcpIsOpen(tcpSocket *socket);
```

```
30 extern int tcpIsClosed(tcpSocket *socket);
31 extern int tcpIsSent(tcpSocket *socket);
32 extern int tcpIsError(tcpSocket *socket);
33
34 // header inspection
35 // ------------------
36 extern int tcpHeaderPorts(tcpMsg msg, UInt16 *srcPort, UInt16 *dstPort);
37
38 // EXTERNAL CALLBACK FUNCTIONS
39 // ----------------------------
40 extern int net_deliver_push(tcpSocket *socket, UInt8 *data, UInt16 dataLen);
41 extern int sendLink(tcpSocket *sock, UInt8 *data, UInt16 dataLen);
```

Again the functional interface on lines 5-32 are self-explanatory, and the other functions follow the scheme presented for the `tcpAPI.h`.

testtcp.c

The TCP evaluation brick, pushes random data on a TCP connection between to ANA nodes. Allows performance recording for a TCP session, and writes these records as *comma-separated values* (CSV) into a file.

A.2.4 Synchronization Service

syncdiscovery.c

The ANA brick implementation for the Sync service, establishing relevant IDPs (Monitoring Framework, PodNet) and taking care of TCP multiplexing / demultiplexing in packet delivery.

syncManager.h

The synchronization manager, keeps a list of available neighbors, and attributes content timestamp to these neighbors. On relevant state changes the all subscribed content dissemination bricks are sent a notification.

A.2.5 Neighbor Discovery

config.h

Neighbor discovery parameters, such as the heartbeat interval and event timeouts (stability, deletion).

neighbordiscovery.h

The ANA brick implementation for the neighbor discovery, establishing relevant IDPs (Monitoring Framework, Network Broadcast).

neighborManager.h

The neighbor discovery functionality. Including the creation of network discovery packets, the tracking of received discovery packets, and the notification of subscribed bricks (connectivity) on relevant state changes, as described in chapter 3.2.1.

A.2.6 ANA Binaries

These binary files are used to run and configure the `minmex` program, `mxconfig` allows loading and unloading plugins (bricks), and the `vlconfig` program is used for configuring the minmex's network interfaces.

A.2.7 Dynamically Loaded ANA Objects

The `.so` plugin files, representing the individual bricks of the ANA framework, can be loaded at runtime by the minmex. The shell script `ana_podnet_init.sh` makes use of this ability. All the listed modules are needed to operate the PodNet on ANA application.

Appendix B

Platform Independence

B.1 Overview

The file listing provided for the OpenEmbedded toolchain is also serving as the index to this appendix.

In order to cross-compile software, first the configuration files must be made available, as explained in appendix B.2.2. Then the compilation environment must be set up, as described in appendix B.2.3. Then the compilation process may be started, launching the OpenEmbedded toolchain, this procedure is described in detail in appendix B.2.4.

Furthermore these dependencies must be satisfied, in order to be able to cross-compile software with OpenEmbedded toolchain:

- git-core
- bitbake
- help2man
- diffstat
- texi2html
- texinfo
- gawk

```
OpenEmbedded/

 -  source-me.txt - B.2.3
 +  angstrom-stable
   -  checksums.ini
   +  cache
   +  cross
   +  deploy
     +  glibc
       +  ipkg
         +  armv5te
           -  ... - B.2.5
   +  staging
   +  stamps
   +  work
 +  build
   -  ...
   +  conf
     -  local.conf - B.2.2
 +  downloads
   -  ... - B.2.2
 +  mamona-stable
   -  checksums.ini
   +  cache
   +  cross
   +  deploy
     +  deb
       +  deb
         -  ... - B.2.5
   +  staging
   +  stamps
   +  work
 +  org.openembedded.stable
   -  ...
   +  classes
     -  ...- B.2.2
   +  conf
     +  build
       -  ...
     +  distro
       -  ...- B.2.2
     +  machine
       -  ...- B.2.2
   +  contrib
     +  ...
   +  files
     -  ...
   +  packages
     +  ...- B.2.2
     +  ana
       -  ana_1.0.0.bb- B.2.2
       +  files
         +  Makefile- B.2.2
     +  images
       -  ...- B.2.2
   +  site
     -  ...
```

B.2 Descriptions

B.2.1 Obtaining the OpenEmbedded Toolchain

Before any configuration or compilation can start, the OpenEmbedded toolchain must be obtained from the online repositories.

```
#create the working directory
user@nb-4957:/$ mkdir /path/to/OpenEmbedded
user@nb-4957:/$ cd /path/to/OpenEmbedded

#download OpenEmbedded
user@nb-4957:/path/to/OpenEmbedded$ git clone git://git.openembedded.net/
    openembedded.git org.openembedded.stable
user@nb-4957:/path/to/OpenEmbedded$ cd org.openembedded.stable
user@nb-4957:/path/to/OpenEmbedded$ git checkout -b org.openembedded.stable
    origin/org.openembedded.stable
user@nb-4957:/path/to/OpenEmbedded$ git pull
user@nb-4957:/path/to/OpenEmbedded$ cd /path/to/OpenEmbedded
user@nb-4957:/path/to/OpenEmbedded$ wget http://www.angstrom-distribution.org
    /files/source-me.txt
user@nb-4957:/path/to/OpenEmbedded$ mkdir build build/conf && cd build/conf
```

B.2.2 Build Configuration

The files that are mandatory for a compilation are the BitBake recipe for PodNet on ANA, and the patches that the system must apply to the source code. Furthermore the top-level compilation settings are located in the `local.conf` file.

local.conf

Describes the overall building process, i.e. the target architecture and the chosen distribution. The file for the Angstrom compilation used for the Sharp SL-C860 is:

```
# Where to store sources
DL_DIR = "/path/to/OpenEmbedded/downloads"

# Which files do we want to parse:
BBFILES := "/path/to/OpenEmbedded/org.openembedded.stable/packages/*/*.bb"
BBMASK = ""

# ccache always overfill $HOME....
CCACHE=""

# What kind of images do we want?
IMAGE_FSTYPES = "jffs2␣tar.gz␣"

# Set TMPDIR instead of defaulting it to $pwd/tmp
TMPDIR = "/path/to/OpenEmbedded/${DISTRO}-stable/"

# Make use of my SMP box
PARALLEL_MAKE="-j4"
BB_NUMBER_THREADS = "2"

# Set the Distro
DISTRO = "angstrom-2007.1"

# 'uclibc' or 'glibc' or 'eglibc'
#ANGSTROM_MODE = "glibc"

MACHINE = "c7x0"
```

Whereas the file to use with the Nokia N810 looks as follows:

```
# Where to store sources
DL_DIR = "/path/to/OpenEmbedded/downloads"

# Which files do we want to parse:
BBFILES := "/path/to/OpenEmbedded//org.openembedded.stable/packages/*/*.bb"
BBMASK = ""

# ccache always overfill $HOME....
CCACHE=""

# What kind of images do we want?
IMAGE_FSTYPES = "jffs2␣tar.gz␣"

# Set TMPDIR instead of defaulting it to $pwd/tmp
TMPDIR = "/path/to/OpenEmbedded//${DISTRO}-stable/"

# Make use of my SMP box
PARALLEL_MAKE="-j4"
BB_NUMBER_THREADS = "2"

```

```
# Set the Distro
DISTRO = "mamona"

# 'uclibc' or 'glibc' or 'eglibc'
#ANGSTROM_MODE = "glibc"

MACHINE = "nokia800"
```

OpenEmbedded ANA Recipe

The BitBake compilation recipe is written in the Python programming language.

```
DESCRIPTION = "Autonomic␣network␣architecture␣(ANA)."
LICENSE = "GPL"
PR = "r0"

DEPENDS="ncurses␣gtk+"

S = "${WORKDIR}/${P}"

do_fetch () {
    mkdir -p ${WORKDIR}/${P}
    cd ${WORKDIR}/${P}
    cp -r /path/to/ana-core/devel/* ${WORKDIR}/${P}/
}

do_compile () {
    oe_runmake
}

do_install () {
    install -d ${D}${bindir}
    install -m 0755 bin/minmex ${D}${bindir}/
    install -m 0755 bin/mxconfig ${D}${bindir}/
    install -m 0755 bin/vlconfig ${D}${bindir}/

    install -m 0755 so/vlink.so ${D}${bindir}/
    install -m 0755 so/cfinder.so ${D}${bindir}/
    install -m 0755 so/eth-vl.so ${D}${bindir}/
    install -m 0755 so/dispatcher.so ${D}${bindir}/
    install -m 0755 so/connectivity.so ${D}${bindir}/
    install -m 0755 so/neighbordiscovery.so ${D}${bindir}/
    install -m 0755 so/testtcp.so ${D}${bindir}/
    install -m 0755 so/syncdiscovery.so ${D}${bindir}/
    install -m 0755 so/podnet.so ${D}${bindir}/

    # /bin/init is on purpose, it is tried after /sbin/init and /etc/init
    install -d ${D}${base_bindir}
    ln -sf ${bindir}/minmex ${D}${base_bindir}/init
    ln -sf ${bindir}/mxconfig ${D}${base_bindir}/init
    ln -sf ${bindir}/vlconfig ${D}${base_bindir}/init

    ln -sf ${bindir}/vlink.so ${D}${base_bindir}/init
    ln -sf ${bindir}/cfinder.so ${D}${base_bindir}/init
    ln -sf ${bindir}/eth-vl.so ${D}${base_bindir}/init
    ln -sf ${bindir}/dispatcher.so ${D}${base_bindir}/init
    ln -sf ${bindir}/connectivity.so ${D}${base_bindir}/init
    ln -sf ${bindir}/neighbordiscovery.so ${D}${base_bindir}/init
    ln -sf ${bindir}/testtcp.so ${D}${base_bindir}/init
    ln -sf ${bindir}/syncdiscovery.so ${D}${base_bindir}/init
    ln -sf ${bindir}/podnet.so ${D}${base_bindir}/init
}
```

OpenEmbedded ANA Patch

In order to reflect differences in Makefiles for different platforms, the original Makefile is patched with following differences. In our case the standard compilers defined in the Makefile are commented out, in order to allow OpenEmbedded to choose the right compiler for the selected scenario.

```
--- Makefile    2009-02-24 15:04:35.000000000 +0100
+++ Makefile    2009-03-26 12:30:40.000000000 +0100
@@ -1,9 +1,9 @@

 # find out what kernel version is running

-CC := gcc
-CXX := g++
-LD := ld
+#CC := gcc
+#CXX := g++
+#LD := ld

 # For cross-compilation, replace CC and LD with your cross compiler and linker
 # CC := /disk2/kamikaze_svn/trunk/staging_dir/toolchain-mipsel_gcc4.2.4/bin/mipsel-linux-uclibc-gcc
```

OpenEmbedded Distro

Contains the BitBake files that describe a software distribution, i.e., the set of packages included in the operating system.

OpenEmbedded Machine

Contains BitBake files that describe the specific hardware architectures, that are supported by OpenEmbedded

OpenEmbedded Packages

Contains all packages that are available for compilation in the OpenEmbedded toolchain.

OpenEmbedded Downloads

Often package sources are directly downloaded from the internet (`do_fetch()` function in the BitBake recipe). The tarballs are placed in this directory, to avoid repetitive downloads of the same packages.

OpenEmbedded Classes

Defines different classes of BitBake recipes, as for example the cpan-class (used with the PERL programming language).

OpenEmbedded Images

Defines the basic building blocks (package sets) for the distributions available in B.2.2. For example a base-image or a x11-image.

B.2.3 Environment Setup

source-me.txt

In order to set up the compilation environment, OpenEmbedded provides an export file, that sets the operating system's environment variables (as e.g. the PATH variable).

B.2.4 Compilation Process

To compile a package, following steps need to be executed.

```
#change the working directory
user@nb-4957:/$ cd /path/to/OpenEmbedded

#setup the environment
root@nb-4957:/path/to/OpenEmbedded$ echo 0 > /proc/sys/vm/mmap_min_addr
user@nb-4957:/path/to/OpenEmbedded$ source source-me.txt
user@nb-4957:/path/to/OpenEmbedded/build$ cd ../org.openembedded.stable/

#update the local version with the online repository
user@nb-4957:/path/to/OpenEmbedded$ git pull --rebase

#clean the working directories
#(if there exists a previous compilation, this step is mandatory)
user@nb-4957:/path/to/OpenEmbedded$ bitbake -c clean ana

#start the compilation process
user@nb-4957:/path/to/OpenEmbedded$ bitbake ana

#if we want to build complete distros
#(use the following to build three different kinds of images)
user@nb-4957:/path/to/OpenEmbedded$ bitbake base-image ; bitbake console-
    image ; bitbake x11-image
```

B.2.5 Package Deployment

Angstrom Deploy

This directory will contain all the IPK software bundles compiled for the Angstrom distribution, targeting the Sharp SL-C860.

Mamona Deploy

This directory will contain all the DEB software bundles compiled for the Mamona distribution, targeting the Nokia N810. Unfortunately the Nokia does not seem able to handle these kind of files, therefore the Nokia Distribution was repackaged as a tarball.