Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

Institut für Technische Informatik und Kommunikationsnetze

Stephan Dudler

autonomic network architecture

New Protocols and Applications for the Future Internet

Master Thesis, MA-2007-39
October 2007 until March 2008

Supervisor: Prof. Dr. Bernhard Plattner
Advisors: Ariane Keller, Theus Hossmann, Dr. Martin May

Abstract

In recent years, the Internet has reached enormous popularity, but at the same time its weaknesses become evident: Huge efforts have to be taken to manage all participating nodes, and the infrastructure is not suitable to integrate emerging network paradigms such as sensor networks or delay tolerant networks. Therefore several research projects try to reinvent the Internet and provide a network architecture which is better armed for future needs.

This Master Thesis is situated in the ANA project which builds an autonomic network based on a clean slate approach. The ultimate goal is to develop a novel autonomic network architecture that enables flexible, dynamic, and fully autonomic formation of networks. In an ANA node two main components can be identified: The MINMEX and the Playground. The Playground hosts the elements (functional blocks) providing networking functionality. The MINMEX ties these functional blocks together and dispatches the data amongst them.

In this Master Thesis two major functional blocks in the ANA Playground have been developed, extending the features of ANA increasingly: The Internet Protocol (IP) together with the Routing Information Protocol (RIP), and field-based service discovery. Due to IP as first use case for a complex network protocol, it is now possible to communicate over disparate Ethernet segments whereas before, messages could only be exchanged within Ethernet segments. In addition, RIP offers the leadoff opportunity to perform routing inside an ANA network. Field-based service discovery as cutting-edge protocol is a combination of field-based routing and publish-subscribe service discovery which provides network-wide publishing and address-agnostic service discovery.

The decomposition of each protocol by this Master Thesis, resulting in a very modular design, exceedingly flexible, as each protocol piece can be exchanged and extended effortlessly.

Acknowledgments

With this Master Thesis I conclude my studies in Information Technology and Electrical Engineering at the Swiss Federal Institute of Technology (ETH) in Zurich.

Without the support of many people this Master Thesis would not have been possible and I wish to enunciate my gratitude to them.

I would like to thank Prof. Dr. Bernhard Plattner for giving me the opportunity to write my Master Thesis at the Communication Systems Group and for the supervision of my work.

Special thanks to my advisors: Dr. Martin May for the confidence and the chance to work in this fascinating and challenging project, Ariane Keller and Theus Hossmann for the time and effort put in supporting me during the last six months. Their suggestions and constructive feedback guided me through the work on this Master Thesis. It was a pleasure collaborating with them. In addition, the trip to Liège (Belgium) was outstanding.

I would like to thank Thomas Steingruber for providing the technical equipment.

I would like to thank Dr. Christophe Jelger and Ghazi Bouabene from the University of Basel for the helpful discussions and explanations during the meetings and progression of this work.

Finally, special thanks to all my friends and colleagues for refreshing my mind by playing numerous hours of foosball and to my parents for their unconditional support and particularly for the possibility to obtain this excellent education. Last but not least, I would like to thank Fabienne Gadient for her emotional solicitousness.

Zurich, March 2008

Stephan Dudler

Contents

List of Figures

List of Tables

Chapter 1

Introduction

The ANA (Autonomic Network Architecture) Project [1] aims at exploring novel ways of organizing and using networks beyond legacy Internet technology. The ultimate goal is to design and develop a novel autonomic network architecture that enables flexible, dynamic, and fully autonomous formation of network nodes as well as whole networks. Universities and research institutes from Europe and Northern America are participating in this project.

1.1 Motivation

Nowadays, the Internet has reached enormous popularity. Mostly due to the enormous amount of applications provided and the ease of use for the end-user. But as the number and diversity of network capable devices increase and the applications make higher demands on the underlying network, some drawbacks get apparent [2].

- For small devices (e.g. in sensor networks) it is not feasible to provide the whole TCP/IP protocol stack, since it consumes a lot of resources. Therefore, it is desirable to let them communicate using a simpler protocol. But unfortunately the Internet requires the IP protocol for every participating node [2, 21].
- With the number of participants in the Internet the management overhead increases. If one system administrator is assumed per 100 nodes, 10 million system administrators are needed for 1 billion nodes. The introduction of ubiquitous computing and sensor networks may increase the number of nodes beyond 1 billion. Therefore system administrators as well as users may be swamped with the increasing system complexity [10].
- The success of the Internet made it a valuable goal for attackers: Viruses, worms and Denial-of-Service attacks are well known problems. The Internet was not designed with security in mind, but as an open system with distributed control and mutual trust [6]. Therefore it is not possible to protect the network against malicious users without changing the architecture of the Internet significantly [7].

- Firewalls as well as NAT[1] enabled routers pose difficulties for some legitimate applications like VoIP[2]. They block incoming connections, either because the specified destination port is blocked, or because the specified port is not yet known by the NAT device. Firewalls may have to be reconfigured, in order to allow incoming traffic to some applications. And nodes behind a NAT device have to initiate all connections. This is difficult in case of VoIP because the caller is never known in advance [2].

- Many applications have some quality of service (QoS) requirements. But the Internet does not provide any help for QoS. In order to provide optimal performance some applications (e.g. video conferencing) need to estimate the available bandwidth, latency etc. If many applications perform these measurements actively (e.g. by sending probe packets), the network load increases unnecessarily [8]. It would be preferable that the network itself provides some QoS mechanisms as well as information about the network status.

To eliminate these drawbacks the core network infrastructure needs to be changed fundamentally. Unfortunately, the infrastructure of the Internet is not likely to change as several examples (RSVP, IPv6 etc.) have shown in the past. One major problem are the Internet Service Providers (ISP) which are very conservative in deploying anything that does not lead to a financial benefit [9]. Reducing the enormous management overhead would be a motivation for ISPs to introduce a new networking architecture. Such a future network architecture should resolve all drawbacks from the current Internet, and it should be flexible enough to integrate new solutions instead of putting new functionality on top of the network architecture [2].

1.2 The ANA Project

The ANA project tries to build an autonomic network based on a clean slate approach. It is a European Union funded project in “Situated and Autonomic Communications” [11]. The ANA project has started in January 2006 and will last until the end of 2009. Universities, research institutes and industry partners from Europe are participating in this project.

The ultimate goal is to develop a novel autonomic network architecture that enables flexible, dynamic and full autonomic formation of network nodes as well as whole networks. It should exhibit a maximum degree of flexibility and provide support for functional scaling. Functional scaling means that the network is able to completely integrate new functionality. This is accomplished by abandoning the one-size-fits-all network architecture of the Internet and by providing an architectural framework which enables the coexistence of different network architectures. As a result, a main abstraction of ANA is the *compartment*. Each compartment can be individually managed and may use a completely different set of communication protocols [2, 3].

In ANA, the network stack is not fixed as in the Internet but it is dynamically built depending on the networks needs. This flexibility is achieved by defining a *Minimal INfrastructure for Maximal EXtensibility (MINMEX)* which has to be provided by any ANA node. The MINMEX provides the functionality which is required to run ANA. The actual networking functionality is implemented in the ANA *Playground*. The Playground is an accumulation of *functional blocks*, each of them provides a certain networking service (e.g. encryption, compression, reliable packet transport etc.). The MINMEX coordinates the packet flow from one functional block to another. For this reason there is no direct communication between different functional blocks but all communication is routed over the MINMEX [2]. For more information about the ANA architecture, refer to chapter 3 or the ANA Blueprint [3].

[1]Network Address Translation [42]

[2]Voice over IP, for example realized by the Session Initiation Protocol (SIP) [43]

1.3 Goals of this Master Thesis

This Master Thesis is situated in the ANA project, more exactly in the ANA Playground. The task of this Master Thesis is to develop new ideas for the Playground part. Hence, new applications and protocols need to be designed, implemented and tested in the existing ANA prototype. Ultimately, this Master Thesis aims at interconnecting all network nodes which calls for addressing and routing. Thus, it covers mainly the following areas:

1. The design, thus the analysis and decomposition as well as the implementation of the Internet Protocol (IP) for ANA as first use case for a complex network protocol. This area also serves as appliance to get familiar with the ANA code and acts as a reference to today's Internet architecture. Additionally, the implementation of the Routing Information Protocol (RIP) as first opportunity to perform routing inside an ANA network is realized.
2. The design and implementation of field-based service discovery for ANA. Field-based service discovery is a combination of field-based routing and publish-subscribe service discovery as exploring novel and cutting-edge protocol in the ANA Playground.

For maximum profit, routing is realized by RIP primarily for wired networks as well as field-based routing predestined for Mobile Ad Hoc Networks (MANET).

Besides these main topics, a large variety of other tasks were performed, reaching from debugging of code in different ANA areas and changing APIs, to participate in meetings and giving a presentation on an ANA coding workshop in Liège, Belgium.

1.3.1 Status of Development at Begin of this Thesis

When this Master Thesis started in October 2007 the development of the ANA core software had already reached a prototype status [2]. Therefore, this Master Thesis can fall back on an implemented ANA prototype containing a full functioning MINMEX, an API (described in detail in section 3.3) and a bootstrapping mechanism allowing different nodes to start to communicate together.

Furthermore, the ANA Playground was already charged with two important functionalities building a very useful groundwork for this Master Thesis:

- The virtual link [4] functionality provides the opportunity to connect multiple ANA nodes. In order to achieve this, one has to explicate two tasks:

  1. Connect the hardware physically.
  2. Configure an identical virtual link ID on two nodes desired to communicate with each other over ANA.

  Every virtual link running in an ANA node is able to receive data sent by another virtual link configured with the same virtual link ID. Thus, the virtual link functionality gives the opportunity to install an arbitrary topology of nodes connected physically.

- The Ethernet functionality offers data exchange between nodes connected by a virtual link. The data is encapsulated with an Ethernet header. Additionally, the Ethernet functionality is able to discover other nodes in the same Ethernet segment. Because the protocol stack is built dynamically in the ANA world, the Ethernet functionality further is able to discover which applications and protocols on the other node want to be reachable through Ethernet. The Ethernet functionality corresponds to the data link layer of the OSI reference model.

1.4 Outline

The documentation of this Master Thesis is structured as follows:

- Chapter 2 provides background information regarding today's Internet architecture in order to introduce terminologies used in this Master Thesis. Moreover, it gives an insight to concepts concerning field-based routing and publish-subscribe service discovery.
- Chapter 3 introduces the concepts underlying the ANA architecture, how these concepts are mapped in the ANA prototype and describes the ANA terminology.
- Chapter 4 explains the design, implementation and validation of IP and RIP in ANA, e.g. documents the first main topic of this Master Thesis.
- Chapter 5 deals with the second main topic of this Master Thesis, respectively. It describes the design, implementation and validation of field-based service discovery for ANA.
- Chapter 6 summarizes the contributions of this Master Thesis and gives an outlook over next steps possible in the ANA development process.

Chapter 2

Background Information

The first main topic of this Master Thesis is concerning about the design and implementation of the Internet Protocol (IP) as well as the Routing Information Protocol (RIP) in the ANA world. Therefore, the first part of this chapter presents background information regarding today's Internet architecture. Foremost, the OSI reference model is explained and recovered in the TCP/IP protocol stack. Then, the specifications of IP as network protocol are described. Section 2.3 finally tops off the information and terminologies of today's Internet by illustrating routing and highlighting the difference between routing and forwarding.

The second part of this chapter gives an introduction into the second main topic of this Master Thesis dealing with field-based service discovery for ANA. First of all, in section 2.5, field-based routing is described and after, in section 2.4, publish-subscribe service discovery is illustrated.

Important for this Master Thesis is first and foremost IP (2.2) used for datagram delivery and addressing issues, RIP (2.3) used for routing inside an IP network (with CIDR) as well as field-based routing (2.5) used for service discovery.

2.1 Internet Architecture

Today's Internet architecture can be described with the TCP/IP architecture as a statical protocol stack. The protocol stack is closely aligned to the OSI reference model. Therefore, this section first introduces the OSI reference model, before specifying the TCP/IP protocol stack.

2.1.1 OSI Reference Model

The OSI (Open Systems Interconnection) reference model [15] is a formal definition by ISO[1] of connecting computers. It specifies the split-up of network functionality in seven individual layers, whereas one or multiple protocols implement the functionality of one specific OSI layer. The OSI model, shown in figure 2.1, is not really a protocol stack, but serves as a reference model for real protocol stacks in network nodes [12].

[1] International Organization for Standardization (ISO) [16]

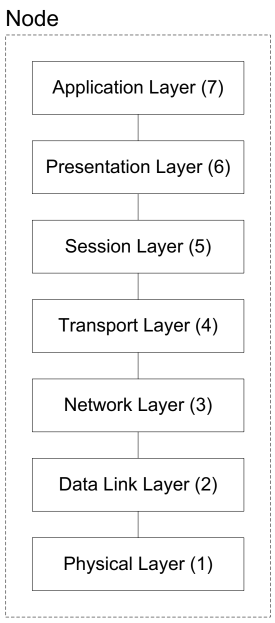


Figure 2.1: The OSI reference model

Hence, it is an abstract description of a communication design. From bottom to top, the seven individual layers of the OSI model may be shortly described as follows:

- **Physical layer (1):** Represents the transmission of individual bits over an interconnection.
- **Data link layer (2):** Arranges the stream of bits into logical sequences.
- **Network layer (3):** Deals with routing between nodes of a packet-switching network, supports fragmentation of packets and reports delivery errors.
- **Transport layer (4):** Controls the reliability of a given link. State and connection oriented transport protocols can demand a retransmission of failed messages.
- **Session layer (5):** Offers the combination of potential different transport streams belonging to one application.
- **Presentation layer (6):** Maps different contexts between application layer entities.
- **Application layer (7):** Provides services to user-defined application processes.

2.1.2 TCP/IP Protocol Stack

Now, the Internet architecture, also called TCP/IP architecture [13], can be considered as a protocol stack related to the OSI reference model. Nevertheless, table 2.1 shows a few variances.

In the TCP/IP architecture, OSI layer 1 and 2 are combined to a network access layer, for example Ethernet[2]. The Ethernet protocol serves as data link protocol arranging bits to Ethernet frames. Additionally, it also defines the physical specifications for devices (e.g. Fast Ethernet or Gigabit Ethernet). The OSI network layer is renamed as Internet layer, because of the Internet Protocol (IP) implementing the network layer functionality. The transport layer remains unaltered and is implemented either by the reliable Transport Control Protocol (TCP) [19] or the unreliable User Datagram Protocol (UDP) [20]. Finally, the session and presentation layer are merged into the application layer acting on top of the TCP/IP architecture.

TCP/IP Architecture	*OSI Layers*	*Examples*
Application	5 – 7	HTTP, FTP, DNS
Transport	4	TCP, UDP
Internet	3	IPv4, IPv6
Network Access	1 – 2	Ethernet, Token Ring

Table 2.1: TCP/IP architecture compared to the OSI reference model [12]

Note that all nodes in today's Internet have to provide the full and statical TCP/IP protocol stack in order to communicate and participate in the Internet [21].

2.2 Internet Protocol

The Internet Protocol (IP) [17] implements the network functionality of the TCP/IP protocol stack. The purpose of IP is to connect different and heterogeneous networks regardless of the lower layers (e.g. Ethernet, Wi-Fi, Token Ring). Therefore, the service model of IP is composed of datagram delivery on the one hand and of unique global addressing amongst network nodes on the other hand [12].

2.2.1 Datagram Delivery

Datagram delivery is an unreliable service, also known as best effort delivery. To achieve this, the IP datagrams are encapsulated with an IP header containing all the information in order that the network is able to forward the datagram. Thus, IP is a data-oriented and connectionless protocol for communicating across a packet-switched internetwork [12]. An IP datagram with an IP header in front looks as depicted in figure 2.2:

[2]IEEE 802.3 Standard

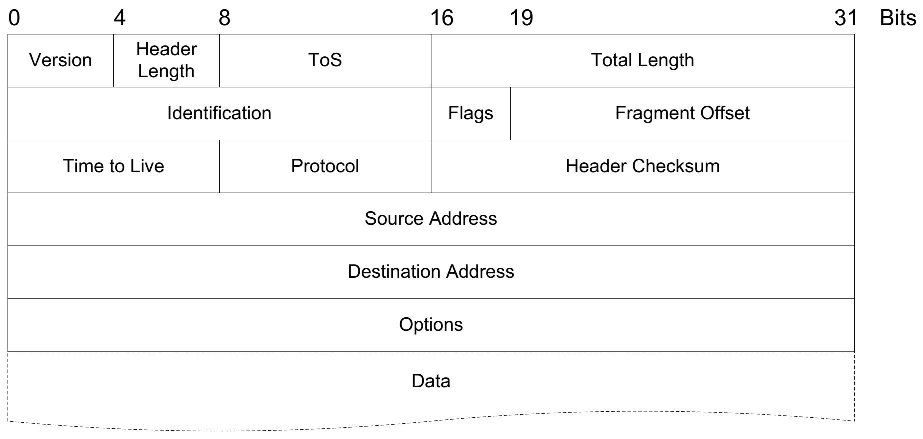


Figure 2.2: The IP header

The IP header consists of 13 fields:

- **Version:** The IP version (e.g. IPv4, IPv6).
- **Header length:** The length of the header telling the number of 32-bit words.
- **ToS:** The Type of Service field can specify a preference for how the datagram should be handled, but this feature is not implemented widely and has been redefined for specific applications (e.g. VoIP).
- **Total length:** Declares the entire datagram size including header and data.
- **Identification:** Identifies IP fragments of an original IP datagram upon fragmentation.
- **Flags:** Used to control fragments. The second out of these three bits is known as the ”Do not fragment” flag and the third one is the ”More fragments” flag (the first bit is reserved and must be zero).
- **Fragment offset:** Specifies the offset of a particular IP fragment relative to the original IP datagram.
- **Time to Live (TTL):** The TTL is a hop count field which will be decremented on each hop. If it reaches zero, the datagram will be discarded.
- **Protocol:** Defines the next protocol in the protocol stack or next header (for example TCP, UDP).
- **Header checksum:** The checksum is used to check delivery errors in the header.
- **Source address:** The IP address of the sender node.
- **Destination address:** The IP address of the receiver node.
- **Options:** This field is for additional information and is the only field which is optional.

The algorithm for checksum computation is declared in RFC 791 [17]: The checksum field is the 16-bit one’s complement of the one’s complement sum of all 16-bit words in the header. For purposes of computing the checksum, the value of the checksum field is zero.

As one can see in the header, IP allows the fragmentation of datagrams. This is necessary when the Maximum Transmission Unit (MTU) of a specific device is smaller than the datagram size. While dividing the datagram into smaller pieces, the needed information for reassembly will be added in the header of each IP fragment. The identification field is needed for the recognition of fragments belonging together, whereas the fragment offset field is used to reassemble in the correct order.

Furthermore, an IP datagram can carry ICMP messages. The Internet Control Message Protocol (ICMP) [23] sits on top of IP in the Internet layer. Its purpose is typically reporting errors in the processing of datagrams. For example, if the TTL of an IP datagram is decremented to zero at a certain router, an ICMP ”Time to live exceeded in transit” message will be generated.

2.2.2 Global Addressing

Providing a unique global addressing amongst network nodes is the second part of the IP service model introduced which is achieved by a well-defined addressing scheme. It should be able to send a datagram to an arbitrary node in a arbitrary network.

Ethernet addresses are globally unique, but they are flat without any structure or localization information for routing processes. Therefore, they can not be used for internetworking [12]. In contrast, IP addresses are hierarchical corresponding to a hierarchy of networks. IPv4 uses 32-bit addresses and the IP address is divided into two parts:

- Network ID
- Host ID

All nodes in the same network contain an identical network ID, whereas the host ID definitely identifies the node in the network. The division of the IP addresses is not identical for each address and obtained by two different schemes:

- Classful
- Classless

Classful networking groups IP addresses in five different classes (A, B, C, D, E) which divide the addresses regarding the length of network and host ID. Table 2.2 pictures out the differences of class A, B and C:

Class	***Network ID***	***Host ID***
A	8-bit	24-bit
B	16-bit	16-bit
C	24-bit	8-bit

Table 2.2: IP address classes A, B and C

This means for example that there exist only 128 different networks of class A or that there are only 128[3] different nodes possible in a class C network. Class D addresses are for multicast purposes and class E is reserved.

[3]including reserved addresses such as broadcast and network address

More important is the second, classless addressing scheme which actually replaces the classful scheme and allows to redivide class A, B and C networks. The amount of IP addresses is delimited and therefore they have to be carefully allocated. With the classful networking only three different classes are possible, but there are a lot of cases imaginable where none of them is suitable. Hence, classless addressing allocates IP addresses more efficiently. This is achieved with subnetting. By means of subnetting, different networks are considered as subnets characterized by a network address on the one hand and a subnet mask on the other hand. A subnet mask is a 32-bit address as well, but consists of a bit string of ”1” with an arbitrary length. Thus, subnet masks divide IP addresses in a much more variable way. For one node this means that it is configured with an IP address and with a subnet mask. In order to retrieve its network address, one has to add up both addresses bit by bit.

The amount of nodes possible in a subnet is defined by the length of the subnet mask. The part for the host ID is the one beyond the bit string of ”1”. However, the last possible IP address is reserved as broadcast address in order to reach all nodes in a subnet and the first possible IP address is reserved as network address, e.g. network ID. Both addresses can not be allocated to nodes.

To overcome the limitation of the number of IP addresses as well, IPv6 [18] as successor of IPv4 redefines the addressing scheme again. In IPv6, 128-bit addresses are used. Based on pessimistic estimates for the efficiency of allocating IPv6 addresses [12], the IPv6 address space features 15’000 addresses for each square meter of earth’s surface which should be really enough for the future.

2.3 Routing and Forwarding

In order to discover desired paths through an internetwork, routing and forwarding are absolutely essential. Routing describes the process of building forwarding tables, whereas forwarding describes the process of looking up a specific destination address in these node local forwarding tables and delivering the data packet to the accurate direction given. Therefore, one has to strongly distinguish between these procedures [12].

As already introduced, the goal of routing is to select paths through an internetwork in order to build a forwarding table for the forwarding process. The routing process can be described as gathering information about the existence of different networks in an internetwork and the corresponding next hops in order to reach them. Thus, routing itself also keeps a table up to date. Actually, the dispartment of the routing table and the forwarding table is a decision of a particular implementation. However, it is recommended to make this difference, because of performance optimization issues and the diversity of interpretation [12, 14]. As mentioned before, routing tables contain in general a mapping of network addresses to next hops, whereas forwarding tables need to contain entries of particular interfaces, thus accurate information about how to forward a data packet to the next hops.

Routing is basically a graph-theoretic problem [12]. One can imagine a graph with nodes representing networks and edges representing interconnections. So, the shortest or the most optimal path between two nodes regarding some kind of metrics for the interconnection has to be selected. Therefore, routing is mostly implemented by routing protocols between nodes as a distributed and dynamical process. In IP, hierarchical network addresses and classless addressing schemes facilitate routing, because blocks of addresses can be grouped together into single routing table entries, also known as Classless Inter-Domain Routing (CIDR). Moreover, there are mainly two well-known classes of routing algorithms:

- Distance vector algorithms
- Link-state algorithms

2.3.1 Distance Vector Algorithms

Distance vector algorithms are based on the concept that each node builds a one-dimensional array or vector containing the distance to all other nodes in the network. The distance is represented by the total cost which is the sum of all metrics values of all interconnections inside a path to another node. Finally, the goal is to select the path to another node with the lowest total cost.

The algorithm[4] operates in a simple manner. Each node sends to its neighbour nodes all vectors containing the destinations, corresponding costs and next hops it knows. Upon reception of such an advertisement, the node compares the received information to what it already knows and updates its routing table with any improvements. Over time, all nodes in the network will discover the next hop for all destinations and the corresponding best total cost.

A famous example for a distance vector routing protocol is the Routing Information Protocol (RIP) [24]. Here, the destinations of desired paths are not nodes, but networks and the metrics is hop count. The advertisements are sent periodically every 30 seconds on the one hand and triggered as soon as the routing table was updated on the other hand. The maximum number of hops allowed with RIP is 15. Furthermore, RIP allows the support of different address families and is not limited to IP addresses [12].

2.3.2 Link-State Algorithms

Link-state algorithms are based on the concept that each node knows its neighbour nodes. When the total amount of these information are distributed to all nodes in the network, each node possesses adequate information to construct a map of the network and thus to build its routing table. Hence, link-state algorithms comprehend two mechanisms. Distributing the information on the one hand and computing the routes out of these information on the other hand [12].

Distributing the information is achieved by flooding. Flooding means that each node will receive a copy of the link-state packet of all nodes. Computing of the routes out of these link-state packets is done by each node independently using a standard shortest path algorithm[5].

A famous example for this routing process is the Open Shortest Path First Protocol (OSPF) [26]. OSPF extends the link-state algorithm described with several improvements such as authentication of routing advertisements, additional hierarchy and load balance.

2.3.3 Inter-Domain Routing

The Internet is organized in autonomous systems (AS). An AS represents an internetwork controlled by an administrative authority, such as a Internet Service Provider (ISP) or a big company [12]. Regarding routing issues, autonomous systems provide an additional routing hierarchy optimizing the scalability of routing protocols. In order to achieve this, routing is furthermore separated into routing inside an AS and between different AS, also called inter-domain routing.

The distance vector and link-state routing protocols described can not be used for inter-domain routing, because they are not scaleable for it. Therefore, distance vector and link-state routing protocols are known as interior gateway protocols (IGP) for intra-domain routing, whereas the exterior gateway protocol (EGP) or its successor the border gateway protocol (BGP) [25] are used for inter-domain routing.

[4]also known as Bellman-Ford algorithm [48]
[5]such as Dijkstra's algorithm [49]

2.4 Publish-Subscribe Service Discovery

The need to discover information is fundamental to any large scale and distributed environment [33]. An upcoming technology is the use of publish-subscribe mechanisms. A publish-subscribe system is a communication infrastructure that enables data access and sharing over disparate systems and among inconsistent data models [34]. Gnutella[6] is an example of a publish-subscribe system.

A publish-subscribe system, also known as pub-sub system, is a routing mechanism that delivers data packets from publishers to interested subscribers [35]. Unlike multicast group communications where group addresses and memberships are statically bound, pub-sub systems use a communication model where the eligibility of group membership is evaluated dynamically. Such a communication system has many potential benefits. For instance, instead of requiring publishers to identify destination addresses for their messages (potentially requiring multiple messages to multiple destinations), a pub-sub network can handle message routing in a way that avoids unnecessary message replications [34].

A pub-sub system further can be seen from a subscriber's point of view. A subscriber who wants to use a certain type of service, but unaware of the existence of this service type or the location of service instances needs a discovery mechanism. Therefore, the intention of a subscriber discovering a specific service type represented by a subscribe message should be automatically guided towards an optimal service instance [27]. This type of view defines a pub-sub system as promising technology for address-agnostic service discovery.

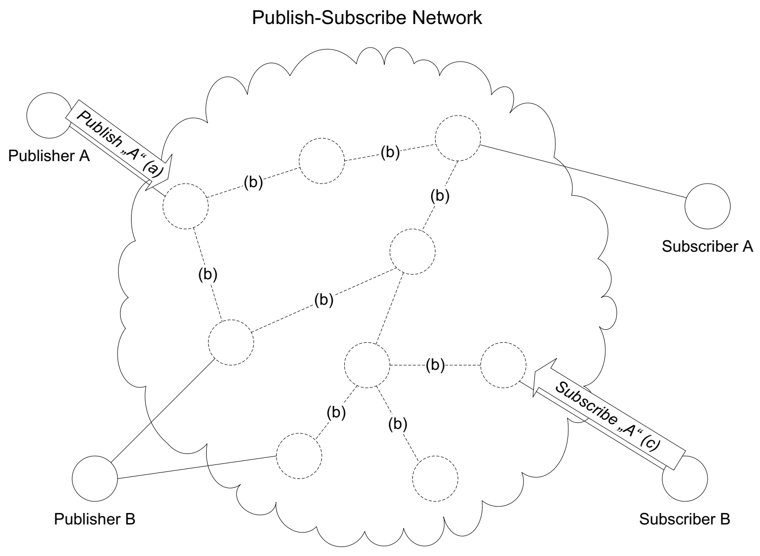


Figure 2.3: Publish-subscribe service discovery

Figure 2.3 shows a pub-sub network and two publishers and subscribers, respectively. Publisher A publishes a certain service "A" in the pub-sub network (a). Now, the information about this service type is distributed and shared in the pub-sub network (b). Thus, the subscribe message from subscriber B who wants to discover service "A" (c) is automatically directed to publisher A by the pub-sub network. So, the pub-sub network is providing the opportunity for decentral and address-agnostic service discovery.

[6]a file sharing network [50]

2.5 Field-Based Routing

Field-based routing is an exploring novel routing mechanism inspired by the field theory from physics. Destination nodes generate a potential field overlay for the network in order that data packets of source nodes modeled as test charges diffuse along the steepest gradient of the potential field [29]. The benefit of field-based routing is that each point in space knows exactly how to participate in the global behaviour based on only the knowledge of its position [30]. Therefore, this approach is very efficient in networks with frequent topology changes and may guarantee loop-free topologies. Field-based routing is very helpful for address-agnostic issues and can be used for MANETs[7], anycast routing or publish-subscribe networks, for example.

Fields are not restricted to one specific destination node. Different nodes can contribute to the same potential field. In that case, the potential field is obtained in the same way as in physics by creating the linear superposition of the influence from the individual nodes of the group. The peaks in the resulting total field represent the location of the destination nodes. Packets following the steepest gradient will then be delivered to any node of this group [32].

This case can be adapted for group members building an entire anycast group. Figure 2.4 depicts the potential field of such an anycast group with ten members.

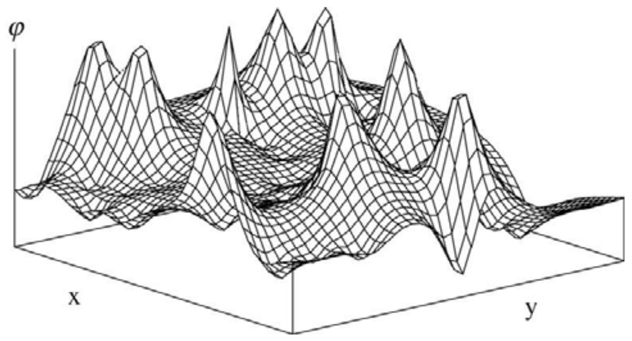


Figure 2.4: Example potential field with ten destination nodes [27]

The potential field for a network performing field-based routing can be computed for example analogously to an electrical potential field in physics [27]. The destination node is related to a point charge Q_j located at $\overrightarrow{r}_j$, whereas data packets are modulated as electrical test charges at position $\overrightarrow{r}$. The resulting potential $\varphi_j(\overrightarrow{r})$ is:

$$\varphi_j(\overrightarrow{r}) = \frac{1}{4\pi\varepsilon} \cdot \frac{Q_j}{|\overrightarrow{r} - \overrightarrow{r}_j|}$$

The fundamental difference between the physical field model and field-based routing in networks is that a field in physics is continuous, whereas in a network the field is only defined at the nodes and propagating over the underlying network links. Therefore, the potential has discrete values and the potential at any node n resulting from this charge is defined as:

$$\varphi_j(n) = c \cdot \frac{Q_j}{dist(n,n_j)}$$

[7]Mobile Ad Hoc Network

The distance between node n and node n_j is represented by $dist(n,n_j)$ and c is a constant. For simplicity, c can be set to $c = 1$ and the distance can be declared as the shortest distance in hops:

$$\varphi_j(n) = \frac{Q_j}{|n-n_j|}$$

Finally, the potential field is a superposition of the individual fields of each destination node in a group:

$$\varphi(n) = \sum_{j=1}^{N} \varphi_j(n) = \sum_{j=1}^{N} \frac{Q_j}{|n-n_j|}$$

Field-based routing can be controlled for example with the point charge Q for point out a specific destination node (by increasing Q) or even more by changing the distance function into an exponential function. Large values for an exponent result in steep potential fields, whereas small values result in flatter distributions.

Finally, routing in this model is realized by forwarding data packets towards the steepest gradient, as already mentioned. This is achieved by comparing the potential values of all neighbour nodes at a node. The steepest gradient is then obtained by the neighbour node with the highest value [27].

Chapter 3

The Architecture of ANA

This chapter introduces the network architecture proposed by the ANA project [1]. The high level design principles such as basic abstractions and building blocks are defined in the ANA Blueprint [3]. The first two sections of the chapter 3.1 and 3.2 describe the basic building blocks and the communication concepts of ANA. For a more detailed discussion, one has to refer to the ANA Blueprint.

The next two sections 3.3 and 3.4 explain fundamental instruments in order to use communication concepts and develop code for ANA. For more detailed instructions regarding development and source code writing in ANA, one has to refer to the ANA core documentation [4].

3.1 Terminology

As already mentioned in the introduction, the ANA project separates the network architecture into two main components:

- MINMEX
- ANA Playground

Actually, there exist a third main component called the hardware abstraction layer. However, it is not relevant for this Master Thesis and therefore not specified more precisely. Nevertheless, these two (three) main components allow basically to run ANA on a physical device (e.g. computer, sensor, network processor, switch, router, etc.). Figure 3.1 shows the collection of these either mandatory or optional software components:

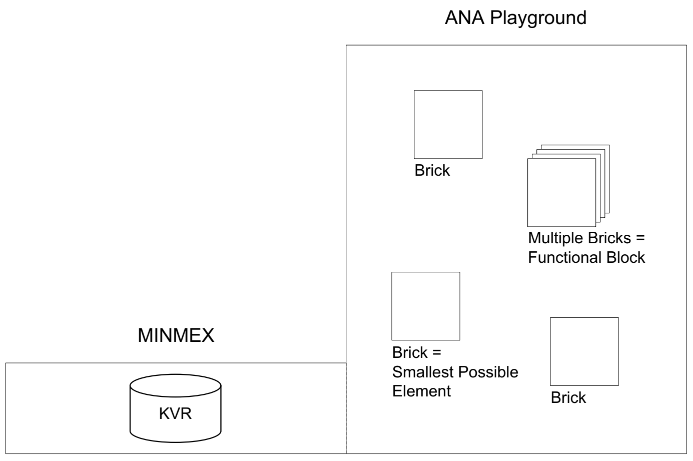

ANA Playground
Brick
Multiple Bricks = Functional Block
Brick = Smallest Possible Element
Brick
MINMEX
KVR

Figure 3.1: MINMEX and Playground building the network architecture

- **MINMEX:** The MINMEX or Minimal INfrastructure for Maximal EXtensibility defines the common denominator among ANA nodes and must be present in all implementations. It provides the basic low level functionality which is required to bootstrap and run ANA.
- **ANA Playground:** The Playground is the execution environment where the more elaborated and complex networking functionality of ANA is placed. The playground hosts the optional protocols and applications that one is free to develop with the help of the functionality provided by the MINMEX elements.

The most important element in the MINMEX concerning this Master Thesis is the following:

- **KVR:** The key-val repository (KVR) is a directory service to access communication mechanisms, protocols and compartments. Functional blocks located in the Playground may add entries with self-chosen keywords which can later be looked up via the standard resolve functionality (explained in section 3.3).

The important elements in the ANA Playground are depicted and related to each other in figure 3.2:

- **Brick:** A brick is the smallest possible element in the ANA Playground providing some functionality. It can be part of a bigger functional block or of a compartment. Together all bricks form the Playground of an ANA node. Before a brick can participate in ANA it has to attach itself to the MINMEX. Bricks interact through the MINMEX. There is no direct communication between two bricks. The MINMEX allows to discover other existing bricks and dispatches messages between IDPs owned by different bricks. The first two bricks in the ANA Playground were the vlink brick and the Ethernet brick as described in the introduction.
- **Functional block:** Functional blocks are the information processing units of ANA. Multiple bricks acting for the same purpose, e.g. a protocol, are combined to a functional block. Functional blocks generate, consume, process or forward information. They might consist of other functional blocks.

To sum it up, a brick is a running instance of an atomic functional block. A functional block is a more abstract concept.

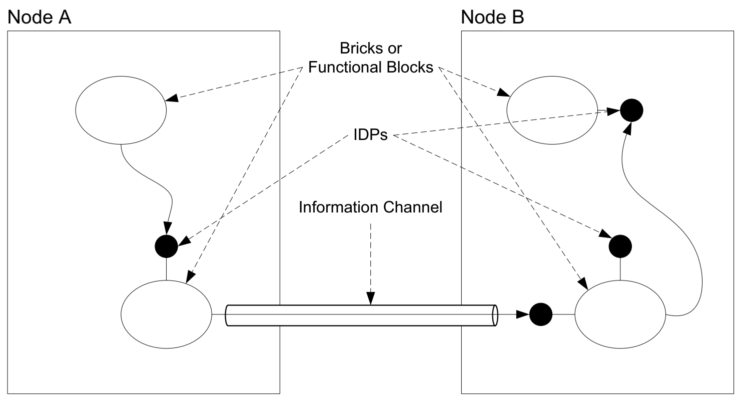


Figure 3.2: Data flow through bricks, functional blocks, IDPs and information channels

- **Information dispatch point (IDP):** IDPs are access points to bricks, functional blocks or information channels. The MINMEX manages the binding of IDPs to bricks. For example, the MINMEX is able to exchange a brick behind a given IDP. This leads to the possibility to exchange functionality, whereas leaving the address constant.
- **Information channel:** Information channels are an abstraction for communication channels. Functional blocks communicate over information channels.
- **Label:** A label is a node local identifier for an IDP.

In order to top off the terminologies, functional blocks might make use of a quick repository (QREP). This is not a concept of the architecture, but an instrument offered by the ANA core software. Similar to the KVR in the MINMEX, the QREP is able to store entries from clients of functional blocks. The QREP is brought up here, because it is used in this Master Thesis and it will be mentioned later in this documentation.

3.2 Compartments

The concept of compartments allows the division of communication networks into smaller units. A compartment is basically a policed set of functional blocks, IDPs and information channels with some commonly agreed set of communication principles, protocols and policies [3]. The boundary of a compartment can be based on technological or administrative boundaries. Compartments using different technologies can communicate together through a common overlay compartment or ”translation” compartment. They provide both: Hiding of compartment internals from the outside world and hiding of communication complexity from its members. Each compartment has to provide the following key functions [2, 3]:

- **Registration and de-registration:** The registration function assures that only admissible communication elements become a member of the compartment. It can be either some kind of authentication function or it can be completely open.
- **Identifier management:** Typically some kind of identifier is assigned to a member during registration phase. A resolution function has to be provided to look up an identifier and to obtain the information necessary to communicate with the requested member.

- **Routing:** The routing function is responsible to select a communication path inside the compartment.

3.2.1 Node Compartment

The node compartment is a special compartment which is present in each ANA node. The MINMEX as well as each functional block residing on one node belong to the node compartment of this node. The node compartment provides mechanisms for functional blocks to make themselves visible for other functional blocks on the same node. It provides also the possibility to search for a functional block which provides a certain functionality. Any functional block or IDP belongs to exactly one node compartment. Therefore the node compartment has full control over the functional blocks and IDPs. The node compartment may provide different views of the actual ANA node to different functional blocks. Thereby, it can hide some functionality from some functional blocks, or define some obligations for certain functional blocks [2].

3.2.2 Network Compartment

A network compartment encompasses several nodes and involves communication across an underlying network infrastructure. Each node which wants to belong to a given network compartment has to implement the required network stack. By providing the network functionality of different compartments, an ANA node may connect to different network compartments. The communication between different nodes is provided by information channels and is typically accomplished over a physical link (but it may also be a virtual one). Network compartments can be combined in a flexible way. A network compartment may combine the functionality of several OSI layers, or it may provide only a part of an OSI layer. To allow communication between different network compartments they may be layered. Therefore, a communication between two nodes may traverse several network compartments [2]. Figure 3.3 shows the layering of different compartments.

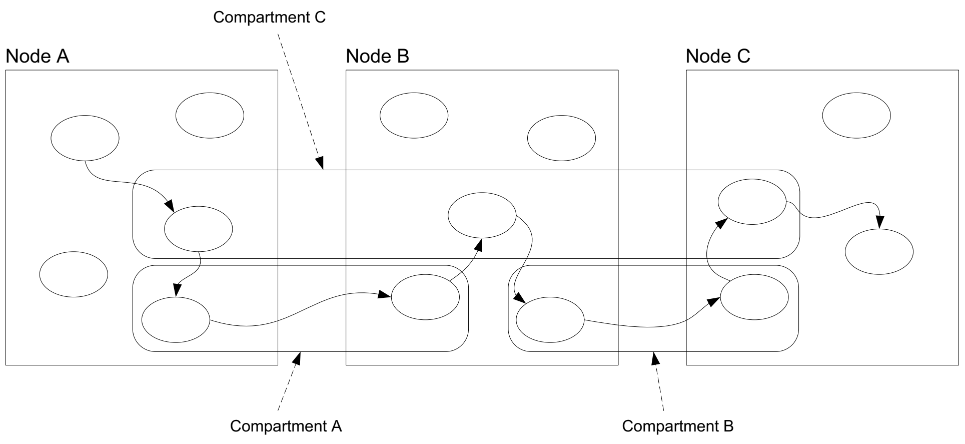


Figure 3.3: Layering of network compartments

3.3 ANA API

This section describes the basic ANA API which is used by bricks to communicate with the MINMEX. The ANA API is divided into different packages or levels that rely on each other [2].

- **API Level -1:** Defines the platform dependent communication mechanism between the bricks and the MINMEX.
- **API Level 0 (AL0):** AL0 groups all functions that permit to access the MINMEX in a platform unspecific way. It consists of attachment and detachment functions as well as a registration function or callback functions.
- **API Level 1 (AL1):** AL1 is a library of procedures for interacting with ANA compartments, information channels and functional blocks. It does not remove the need to manage communication with the MINMEX at AL0. But once attached, there are procedures for creating requests and parsing replays.
- **API Level 2 (AL2):** At AL2, functionality can be accessed through ordinary method calls.

These levels express different degrees of sophistication an do not imply a layering as in networking. If an ANA application wishes, it could be fully based on level 0 [3].

Each compartment is asked to understand some very fundamental and specific API functions belonging to AL2 which represents high level concepts of the ANA network architecture:

- Publish / Unpublish
- Resolve / Lookup

3.3.1 Publish / Unpublish

The publish function is used by an entity (typically a functional block) in order to ask a compartment that it wants to be reachable via this compartment [36]. For instance, a functional block of IP could ask an Ethernet compartment to become reachable via the Ethernet compartment. The publish function provides the opportunity to build a protocol stack dynamically. Figure 3.4 illustrates an example with IP and Ethernet.

In the example, the functional block of IP publishes its IP address in the Ethernet compartment. Hence, the functional block of Ethernet now knows that IP exists and wants to be reachable by the name ”1.2.3.4”. Note that the name ”1.2.3.4” has no meaning within the Ethernet compartment. Additionally, IP tells Ethernet on which IDP it wants to receive further data in order that Ethernet is able to map the name ”1.2.3.4” with the IDP ”i”.

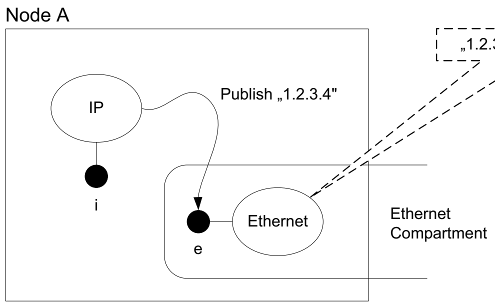


Figure 3.4: Publish example

The publish function prototype at AL2 looks as follows [4]:

```
anaLabel_t anaL2_publish(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *description, int descriptionLength,
AL2Callback_t service, int separateThread);
```

Where:

- `COMPARTMENT_LABEL`: Is the IDP of the compartment functional block to whom the publish is directed.
- `context`: Defines a subset of the compartment in which we want to publish the IDP.
- `contextLength`: Is the size in bytes of the field pointed by `context`.
- `description`: Is the description or name of the service provided by the requester entity.
- `descriptionLength`: Is the size in bytes of the field pointed by `description`.
- `service`: Is the callback function providing the service one wants to publish.
- `separateThread`: Is a flag indicating whether the callback function is supposed to be launched in a separate thread or not.

The unpublish function is the reverse of the publish function and retracts a previous publish activity. The prototype looks as follows [4]:

```
int anaL2_unpublish(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *description, int descriptionLength);
```

The meanings of the arguments are identical as for the publish.

3.3.2 Resolve / Lookup

The resolve function now offers the ability to discover the entities published previously. In other words, an entity asks the compartment, if another entity is reachable via this compartment [36]. Furthermore, if the resolved entity is reachable via the compartment, a new IDP is created and mapped to the IDP previously published by the resolved entity. Hence, after having resolved an entity successfully, one is able to start communicate via an IDP opening an information channel to the resolved entity. Figure 3.5 shows again an example with IP and Ethernet.

In the example, IP B published its name ”1.2.3.4” in the Ethernet compartment. Now, IP A wants to resolve ”1.2.3.4” in the Ethernet compartment and receives a positive reply, because the compartment knows ”1.2.3.4”. Additionally, the reply contains a newly created IDP ”s” opening an information channel to ”1.2.3.4”.

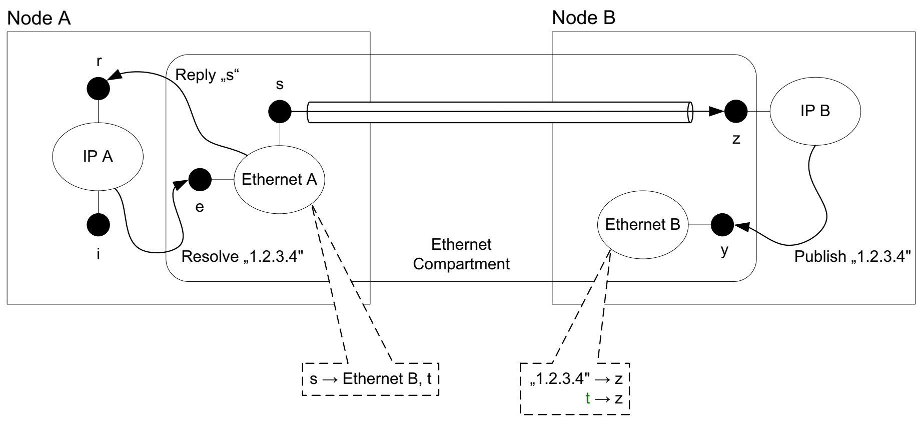


Figure 3.5: Resolve example

The resolve function prototype at AL2 looks as follows [4]:

```
anaLabel_t anaL2_resolve(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *target, int targetLength,
char chanType, void *description, int descriptionLength);
```

Where:

- `COMPARTMENT_LABEL`: Is the IDP of the compartment functional block to whom the resolve request is directed.
- `context`: Delimits a subset of the compartment in which to look for the entity. There are two special types of context defined:
  - ”.” for a node local resolve and
  - ”*” for the biggest subset possible, whereas it is up to the compartment how to implement ”*”.
- `contextLength`: Is the size in bytes of the field pointed by `context`.
- `target`: Is the description or name of the entity one wants to resolve.
- `targetLength`: Is the size in bytes of the field pointed by `target`.
- `chanType`: Determines the type of channel to the entity(es). The possible channel types are ’u’ for unicast, ’a’ for anycast, ’c’ for concast, ’m’ for multicast and ’b’ for broadcast. Indeed, in case the resolve request matches different entities in the compartment, the channel type will indicate to the compartment what kind of channel to build.
- `description`: Is an optional argument and corresponds to the description or name of the resolver entity. This is useful for the resolved entities in order to reach back the resolver.

- `descriptionLength`: Is the size in bytes of the field pointed by `description`.

How the resolve resolution is performed is compartment specific. The example of Ethernet implements resolving by sending a broadcast message and thus, it is analogous to today's ARP (Address Resolution Protocol [37]) request.

The lookup function is very similar to the resolve function, except that the resolve function creates an IDP and furthermore an information channel, whereas the lookup function only returns some reachability information. The prototype of the lookup function looks as follows [4]:

```
int anaL2_lookup(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *target, int targetLength,
struct anaL2_lkpResponse **result, void *description, int descriptionLength);
```

Where all meanings of the arguments are identical, except that `result` is the data structure which will contain the desired reachability information upon lookup reply.

3.4 XRP Messaging

XRP (eXtensible Routing Protocol) defines a common message format in the ANA world. An ANA developer is not forced to use XRP to encode messages, but an entire XRP engine is already implemented and given by the ANA core software and provides a very simple way to encode and decode information. Additionally, the ANA API starting from Level 1 also uses XRP messages to exchange information concerning the API functions.

An XRP message always starts with a command which is followed by a set of classes [4]. The detailed encoding of an XRP message is depicted in figure 3.6:

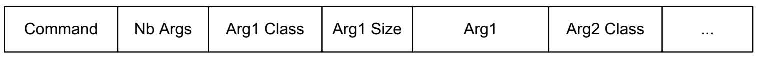


Figure 3.6: XRP message format

- **Command:** Determines the purpose of a message.
- **Nb args:** Is the number of XRP arguments attached to this command.
- **Arg1 class:** Is the XRP class of the first argument attached to this command. By class, a known description (e.g. meta data) about the nature of the argument is meant.
- **Arg1 size:** The size in bytes of the first argument value.
- **Arg1:** Is the value of the first argument associated to the XRP command.

Afterwards, the pattern is repeated for all attached arguments.

The ANA core software also offers a wide XRP API for encoding and decoding XRP messages as well as comparing commands and comparing classes, extracting the number of arguments and many more.

3.5 Summary

This chapter presented the architecture of the ANA project. In each ANA node there are two main components: On the one hand there is the MINMEX which is the minimal component to be provided from each ANA node and on the other hand there is the Playground in which the real functionality provided by the functional blocks is hosted. The smallest possible element of a functional block is a brick. IDPs are access points to bricks, functional blocks or information channels and the information channel itself is an abstraction of a communication channel.

The terminology of compartments is also introduced which allows different network types to coexist. High level concepts such as publish in a compartment in order to be reachable through this compartment or resolve in a compartment in order to discover a desired target and open an information channel impressively shows the ability of ANA building a protocol stack dynamically.

In addition, different API levels offer the possibility to execute such high level functions and the XRP API provides an instrument for ordinary information encoding and decoding.

Chapter 4

IP and Routing in ANA

This chapter describes in detail the implementation of the Internet Protocol (IP) and Routing Information Protocol (RIP) in ANA which enormously extends the features of an ANA network. By means of the IP compartment, it is possible to send messages over multiple Ethernet segments, whereas without, only messages inside the same Ethernet segment is possible for communication. Furthermore, RIP offers the leadoff opportunity to perform routing inside an ANA network. IP and RIP was introduced in theory in section 2.2 and 2.3.

The first section of this chapter covers the design of the IP and RIP architecture for ANA. It gives an overview and comparison to the standard IP as well as specifies the analysis and decomposition in order to fit into the modular and flexible ANA world. Section 4.2 details the precise implementation of every brick involved and defines brick and node communication interfaces. The IP and RIP compartment finally is validated in section 4.3 and summarized in the last section of this chapter.

4.1 Design

This section explains the design of the IP and RIP architecture and pictures out the differences to the standard IP. First and foremost, it particularizes the decomposition of IP in order to provide full flexibility and extensibility.

4.1.1 Protocol Overview

IP, introduced in section 2.2, acts as network protocol in the network layer. The purpose of IP is to connect different networks and provide unique global addressing. This is achieved by encapsulating the data packets with an IP header and the usage of a well-defined addressing scheme.

This Master Thesis decomposes IP in order to abstract the needed and most important pieces for ANA. As can be seen in table 4.1, IP in ANA is therefore different to the standard IP [17] and does not offer the full functionality. Naturally, encapsulation of the datagrams is fundamental, hence the IP header is supported and of course looks identical to the standard IP header. Nevertheless, certain fields of the IP header in ANA have to be detailed specially, as explained in the implementation section 4.2. Furthermore, checksum computation is supported as well as the original addressing scheme. Thus, the premises for connecting different networks and for sending messages between IP subnets are given and the ability to forward IP datagrams is supported accurately. However, some functionality such as fragmentation, IP options and ICMP is not mandatory for this Master Thesis and therefore not included.

Interesting to know, in ANA each node is a router and therefore always tries to forward foreign IP datagrams to a suitable interface.

Supported functionality	***Not supported functionality***
IP header and encapsulation	Fragmentation
Checksum computation	IP options
Addressing and forwarding	ICMP messages

Table 4.1: Differences between IP in ANA and the standard IP

In addition, ANA treats the protocol field or next header field of the IP header in a different way compared to the standard architecture. Because the protocol stack is built dynamically ANA does not know fixed next header codes in the next header field as the standard IP does [22]. Each protocol or user application which wants to be reachable through IP first publishes itself in the IP compartment (refer to 3.3). The next header code corresponding to this user application will be generated dynamically during the publish activity, i.e. randomly. Hence, the next header codes can not be known initially. If a user application intends to communicate with another user application it first has to ask the compartment in order to retrieve the corresponding next header code. This is exactly what the resolve functionality, illustrated in 3.3, is for.

Furthermore, there are information channels like multicast or broadcast channels which aim at multiple targets. However, the next header codes of these targets are not identical as a result of the dynamic and random allocation. Obviously, it is unlikely to retrieve all next header codes even more if the amount of targets or addresses are not known in advance. Therefore, ANA uses two ways to deliver IP datagrams:

- For information channels aiming at one specific target a resolve request is processed by the IP compartment in order to discover the desired next header code. An IDP is created which leads to encapsulating payload with an IP header containing the resolved next header code.
- For information channels aiming at multiple targets a resolve request creates an IDP which leads to encapsulating payload with an IP header containing a special delivery flag in the next header field but a description for the targets in the payload. Therefore, the special flag mentioned indicates the compartment to lookup the targets by means of a description instead of a next header code.

IP datagrams containing resolve requests are indicated by a special flag in the next header field as well. Refer to the implementation section 4.2 for a more detailed discussion about these flags.

RIP, introduced in section 2.3, operates as routing protocol in order to afford the aspired forwarding feature. Remember, routing and forwarding are two different processes but usually, forwarding is done by dint of the results of routing. Besides certain meanderings concerning the implementation (4.2), RIP in ANA provides all the features of the standard RIP [24].

4.1.2 Architecture and Decomposition

In order to design an IP architecture to provide full flexibility and extensibility, one has to analyze and separate the different main pieces of IP. Depicted in figure 4.1, the decomposition of IP and RIP results in a very flexible and modular architecture. The main pieces of the protocol are independent from each other and individually exchangeable and extensible.

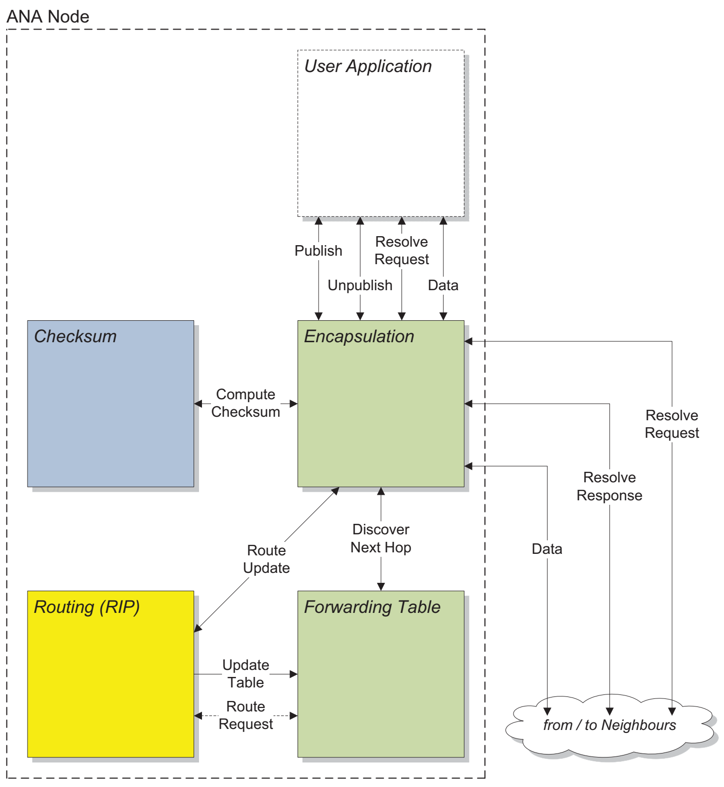


Figure 4.1: Node overview of the IP and RIP architecture

This Master Thesis strongly separates between the mentioned four different processes encapsulation, checksum computation, forwarding and routing.

- The encapsulation process handles, as the name implies, the encapsulation of payload with the IP header. Moreover, it distributes the IP datagrams in both directions: Receiving and transmitting.
- Computing the checksum is rolled out from the encapsulation process to give the opportunity to be reused from different other processes. Thus, the checksum computation process is responsible for computing the correct checksums of the IP headers exchanged with the encapsulation process.
- Forwarding is achieved with help of the forwarding table, sourced out as well. Therefore the encapsulation process looks up destination IP addresses in the forwarding table in order to perform the forwarding process.
- Finally, routing is a separate process which updates the forwarding table continuously. As mentioned before, this Master Thesis uses RIP in order to perform leadoff routing between IP nodes in an ANA network.

The benefits of this modular and flexible architecture are perspicuous. If one wants to change the routing process, simply the routing process has to be exchanged and all other processes remain unaltered. If one needs to extend the whole protocol stack and develops, as an example, a TCP layer, the separated checksum process can be reused, because the algorithm is identical. In conclusion, the design and modular architecture segmented by this Master Thesis offers full flexibility and extensibility as aspired by the ANA project.

For the purpose of providing the described flexibility, well-defined communication interfaces between the individual processes are needed. The next part explains the messages between the processes and between the nodes in order to make up the functionality of IP and RIP in ANA, whereas the concrete and precise message formats can be found in the implementation section 4.2.

4.1.3 Communication Interfaces

The described architecture needs specific message exchange between the individual processes in order to function properly. Furthermore, the format of the message has to be well-defined to achieve the aspired flexibility. This part describes all the messages existing by design and shown in figure 4.1, whereas the implementation section 4.2 deals with the accurate implementation of the interfaces. The unique message names introduced in this part are used throughout the rest of this documentation. Moreover, communication inside a node and communication between nodes is differed.

4.1.3.1 Inside a Node

- Messages received by the encapsulation process:

  This is the interface between a user application and the IP compartment. If one develops a user application of IP he has to use this interface.

  - **PUBLISH:** Publish messages are sent by user applications which want to be visible in the ANA network and reachable through IP. One can find more details in section 3.3.
  - **UNPUBLISH:** Unpublish messages are the opposite of publish messages. A user application is able to retract the previous publishing. This is described in detail in section 3.3.
  - **RESOLVE:** A resolve message is used by user applications to setup an information channel to another user application. As explained in section 3.3, this activity has to precede data messages.
  - **Data:** After having resolved a target successfully and opened an information channel, a user application can start to communicate with data messages (payload) to newly created IDPs by the IP compartment.

- Messages received by the checksum computing process:

  This is the interface between the encapsulation and the checksum computing algorithm [17].

  - **DATA:** A data message for the checksum computing process simply contains the IP header. The checksum computing process will compute the checksum and send it back.

- Messages received by the forwarding table:

  This is the interface between encapsulation and the forwarding table as well as between routing and the forwarding table.

  - **DISCOVER:** A discover message contains the destination IP address for which the correct next hop has to be looked up in the forwarding table in order to forward the IP datagram.
  - **CHANGE:** A change message updates the forwarding table. Therefore it is coming from a routing process and contains the desired routing information.
  - **ROUTEREQ:** The routing table of the routing process is empty in the beginning. Even the subnets in which the node is located itself are missing. The routing process can ask the forwarding table which has been configured before with these interesting information. Hence, this kind of message triggers the dispatch of all entries in the forwarding table for a routing process which wants to know initially all the attached subnets of the node.

- Messages received by the routing process:

  These messages characterize no additional interface but describes the routing advertisements for the routing process.

  - **ROUTEUP:** This message type denotes route updates which have their seeds in routing processes on remote nodes.

4.1.3.2 Between Nodes

- Messages received by the encapsulation process:

  The encapsulation process is the only process which is able to receive and understand genuine IP datagrams from neighbour nodes. There are different kind of IP datagrams existing in the compartment. On one hand, there are IP datagrams processing a resolve request from a user application and on the other hand, there are IP datagrams containing the actual payload for user applications.

  - **resolve request:** A resolve request message is the result of a *RESOLVE* message from a user application. It wants to discover the desired target (e.g. next header code) on a remote node in order to open an information channel. The information channel therefore is of type 'u' or 'a'. Refer to the implementation section 4.2 for a more detailed discussion.
  - **resolve response:** A resolve response represents the successful discovery of a resolved target on a remote node. An information channel will be opened and the corresponding IDP is sent back to the origin user application.
  - **data:** As the name already implies, this kind of message type indicates data messages (payload).

4.2 Implementation

This section describes the precise coding of the IP and RIP architecture in the ANA Playground. It shows how it becomes part of the dynamic protocol stack, how the messages of a communication interfaces are encoded and pictures out the development of each brick accounting for the functional block of IP and RIP.

4.2.1 Overview

IP and RIP is implemented in the ANA Playground as first use case for a complex protocol. The protocol is represented by a functional block. Each process separated and explained in the design section 4.1 is implemented as an individual brick whereas all bricks together form the mentioned functional block. The messages exchanged between the bricks are encoded as XRP messages, except the data message (payload). XRP is introduced in 3.4. The messages exchanged between the nodes are genuine IP datagrams.

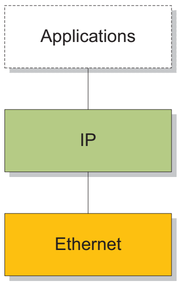


Figure 4.2: Protocol stack of IP running on top of Ethernet

Noticeable, the IP and RIP compartment needs another compartment in order to send the data packets to the wire and identifying the neighbour nodes. For now, the Ethernet compartment and thus the Ethernet brick [4] is used as data link compartment, as depicted in figure 4.2. Therefore, the IP address along with an IDP to the encapsulation brick (ip_enc.c, figure 4.3) which is communicating with neighbour nodes is published in the Ethernet brick. This behaviour is indicating the establishment of a dynamic protocol stack. Mentionable, there are also other data link compartments imaginable, or even more, the IP compartment directly attaches to the vlink brick [4] of the ANA node. This is achieved thanks to the flexible and extensible architecture of ANA as well as the architecture of IP in ANA designed by this Master Thesis. More on this can be found in section 6.2 about further work.

The ANA API functions supported by this compartment are publish, unpublish and resolve functions:

```
anaLabel_t anaL2_publish(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *description, int descriptionLength,
AL2Callback_t service, int separateThread);

int anaL2_unpublish(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *description, int descriptionLength);

anaLabel_t anaL2_resolve(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *target, int targetLength,
char chanType, void *description, int descriptionLength);
```

Publish and unpublish is as explained in 3.3 for user applications which want to be reachable through IP. The resolve function (3.3) is necessary in order to open an information channel to a desired target, before one is able to send data messages. In the IP compartment the `context` argument, thus the scope of the resolve function is implemented as follows:

- "." / "localhost" or "127.0.0.1" in order to reach only local targets.
- An arbitrary IP address, e.g. "10.0.1.2" to reach targets on a specific node.
- The address of a subnet, e.g. "10.0.1.255" in order to reach targets on all nodes in a specific subnet.
- "*" or "255.255.255.255" in order to reach targets on nodes inside all attached subnets. Note that in the ANA world sending to "255.255.255.255" is a reasonable activity resulting in the summation of broadcasts on all attached subnets.

The `chanType` argument is implemented as follows:

- 'b' or 'm' in order to reach multiple targets (broadcast channel, no guarantee about the existence of the targets).
- 'u' or 'a' in order to reach exactly one target (unicast channel, the target actually responds to the resolve process).

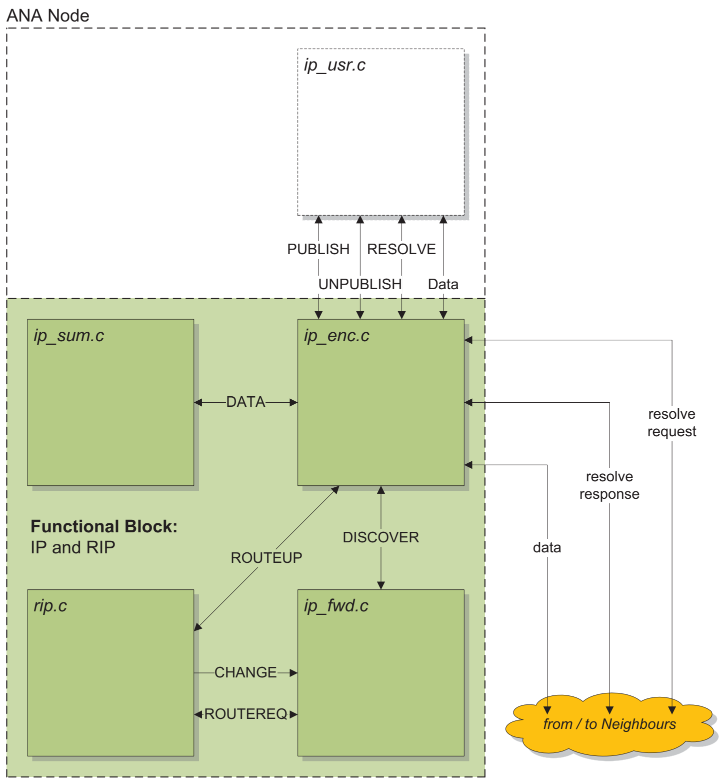


Figure 4.3: Functional block of IP and RIP in the ANA Playground

Figure 4.3 shows the functional block of IP and RIP in the ANA Playground. It is on purpose a similar illustration to the one in the design section 4.1. This time, the separated processes are displayed as the actual bricks and the communication is substituted with the message types introduced in the design section 4.1.

The messages between nodes are, as mentioned before, regular IP datagrams. Nevertheless, one has to notice the following aberrations and assumptions by this Master Thesis concerning the IP header fields:

- **Version:** The IP version is set constantly to ”4”.
- **Header length:** Because there are no IP options, the header length is set constantly to ”5”.
- **ToS:** The ToS field is not used yet and thus, set constantly to zero.
- **Identification:** Because there is no fragmentation supported, this field is set constantly to zero.
- **Flags and Fragment offset:** No fragmentation leads the ”Do not fragment” flag set to ”1” and the rest set to zero.
- **TTL:** The TTL is set initially to the default TTL of ”64”.
- **Options:** No IP options.

Furthermore, a small aberration of the implementation of RIP in ANA compared to the standard RIP can be observed. This Master Thesis implemented RIP as a user application for the IP compartment. Therefore, the advertisements are broadcasted and received over IP whereas in the standard RIP, the advertisements are sent over UDP and with special IP multicast addresses. UDP actually does not bring any additional functionality. In addition, there is no standard RIP header existing because the advertisements are XRP encoded, exploiting the predestined container functionality of XRP.

4.2.2 Configuration

In order to configure the ANA compartment, an additional, special brick was developed in this Master Thesis. The configuration brick acts similar to the well-known `ifconfig` and `route` commands. The brick is indicated as special because after finishing its job, it closes automatically which is in contrast to the other bricks.

There are three different actions possible with the configuration brick: Bind a specific IP address to a specific interface (1). Define a default gateway for a specific ANA node (2) and manually add new routes to the forwarding table (3).

Figure 4.4 shows the processing of the configuration brick:

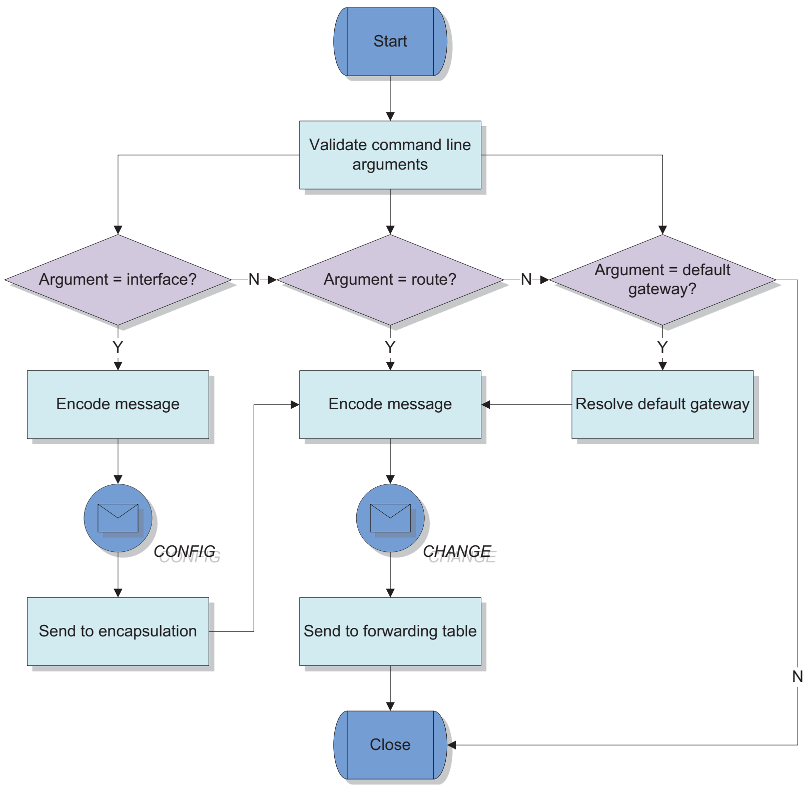


Figure 4.4: Configuration process for an ANA node running IP

1. Configuring an IP address:
   (a) A *CONFIG* message is encoded and sent to the encapsulation brick in order to bind the given IP address with an Ethernet brick.
   (b) A *CHANGE* message is encoded and sent to the forwarding table in order to add the given subnet.

   The configuration is started with the following command line arguments:

   ```
   $ ./bin/ipconfig -n MINMEX_CONTROL_GATE -i IP_ADDRESS -m SUBNET_MASK
   -e NAME_OF_ETHERNET_BRICK_IN_KVR
   ```

2. Configuring a default gateway:
   (a) The brick resolves the default gateway in order to know in which subnet it is located.
   (b) A *CHANGE* message is encoded and sent to the forwarding table in order to add the given default gateway.

   The configuration is started with the following command line arguments:

   ```
   $ ./bin/ipconfig -n MINMEX_CONTROL_GATE -g DEFAULT_GATEWAY
   ```

3. Configuring a route:

   (a) A *CHANGE* message is encoded and sent to the forwarding table in order to add the given route containing the subnet, next hop and corresponding interface, e.g. Ethernet brick.

   The configuration is started with the following command line arguments:

   ```
   $ ./bin/ipconfig -n MINMEX_CONTROL_GATE -a SUBNET -a SUBNET_MASK
   -a NEXT_HOP -a NAME_OF_ETHERNET_BRICK_IN_KVR
   ```

Note that it is necessary to configure the IP compartment. Without configuring an IP address for an ANA node, it will not be able to participate in the compartment. For now, this configuration is done manually, however, there are imaginable scenarios to extend this process for an automatic configuration and addressing, similar to Zeroconf (Zero Configuration Networking [41]) as an example. This is possible thanks to the flexible and extensible design of the ANA architecture as well as the architecture of IP in ANA designed by this Master Thesis. More on this is written in section 6.2.

4.2.3 Parametrization

The implementation of IP in ANA offers some ability to tune different parameters and constants. The parameters and constants all are defined in the header file located at `C/include/ip.h` in the trunk directory. The header file is included in all bricks involved.

```
#define IP_CMD 0x96
#define IP_DLV 0x97
#define IP_RES 0x98
#define IP_RSP 0x99
#define RES_ID_SIZE 4
#define MAX_INTERFACES 10
```

The description and effect of each constant is described in this part. The values offered are default values which have been tested.

- **IP_CMD:** This constant is a special next header code for the IP header as described in the design section 4.1. It serves as a flag in order to tell the compartment that this IP datagram can have two special meanings (depending on the first byte in the payload):
  - On the one hand the IP_CMD next header means that this IP datagram handles resolve processes, therefore, is a *resolve request* or *resolve response* message.
  - On the other hand it means that this IP datagram delivers some payload for a target not to identify by the next header code but by the description carried within (e.g. uses a multicast or broadcast information channel).
- **IP_DLV:** This constant always comes together with an IP_CMD datagram. It means that this IP datagram is a broadcast *data* message delivering some payload for a target not to identify by the next header code but by the description carried within. It is the first byte of the payload.
- **IP_RES:** This constant always comes together with an IP_CMD datagram. It means that this IP datagram is a *resolve request* message. It is the first byte of the payload.
- **IP_RSP:** This constant always comes together with an IP_CMD datagram as well. It means that this IP datagram is a *resolve response* message. Again, it is the first byte of the payload.

- **RES_ID_SIZE:** This size in bytes defines the size of the resolve ID identifying different resolve process.
- **MAX_INTERFACES:** This parameter stands for the maximum amount of network interfaces, e.g. Ethernet bricks, for one node. This value is necessary because at a certain point in the program source code, every interface has to be addressed separately.

For the other specifications and data structures which one can find in the header file, refer to the appendix C containing the doxygen documentation of the program source code.

Additionally, the implementation of RIP in ANA offers some parameters as well. They are defined directly in the RIP brick.

```
#define RIP_UP_TIME 30
#define RIP_DEL_TIME 3
```

- **RIP_UP_TIME:** Defines the time in seconds between periodic *ROUTEUP* messages.
- **RIP_DEL_TIME:** This parameter defines the starting value for the age of an entry in the routing table, regarding garbage collecting issues.

4.2.4 Communication Interfaces

This part describes the important interfaces between the individual bricks. Without a clear definition of the interfaces, the architecture can not provide the aspired flexibility. Therefore, one has to picture out the message format of the exchanged information. Messages inside a node are XRP encoded (with the data message from a user application after a successful resolve process as one exception). Messages between nodes are genuine IP datagrams. The following explanations want to picture out the XRP message encoding in detail (XRP commands and XRP classes as introduced in section 3.4). Additionally, the IP datagrams between nodes are disclosed. Obviously, this part is fundamental in order to understand and use the interfaces inside and between nodes.

4.2.4.1 Inside a Node

- Figure 4.5 shows the XRP messages received by the encapsulation brick:

  This is the interface between user application and IP compartment. If one develops a user application of IP he has to use this interface.

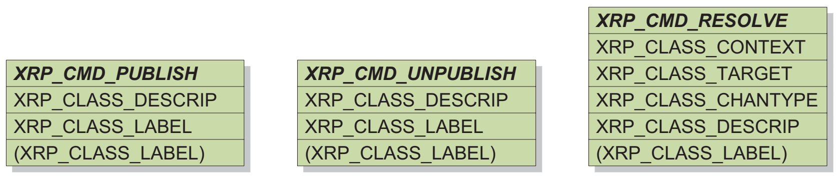


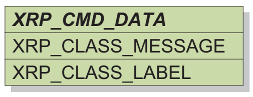
Figure 4.5: XRP messages received by the encapsulation brick

- The *PUBLISH* message is encoded as follows:
  - **XRP_CMD_PUBLISH:** Identifies a *PUBLISH* message.
  - **XRP_CLASS_DESCRIP:** Contains the description of the user application publishing itself.
  - **XRP_CLASS_LABEL:** Contains the data receive IDP in order to send data to the user application publishing itself.
  - **(XRP_CLASS_LABEL):** Contains another IDP which can be used to send back a confirmation of the publish process. This class is optional.
- The *UNPUBLISH* message is encoded as follows:
  - **XRP_CMD_UNPUBLISH:** Identifies an *UNPUBLISH* message.
  - **XRP_CLASS_DESCRIP:** Contains the same description of the user application published before.
  - **XRP_CLASS_LABEL:** Contains the data receive IDP published before.
  - **(XRP_CLASS_LABEL):** Contains another IDP which can be used to send back a confirmation of the unpublish process. This designation is optional.
- The *RESOLVE* message is encoded as follows:
  - **XRP_CMD_RESOLVE:** Identifies a *RESOLVE* message.
  - **XRP_CLASS_CONTEXT:** Contains the context, e.g. the desired scope for this *RESOLVE* message. More information one can find in section 3.3.
  - **XRP_CLASS_TARGET:** Contains the target description.
  - **XRP_CLASS_CHANTYPE:** Contains the channel type in order to distinguish between different resolve behaviours. Again, more information is written in section 3.3 and in the realization part.
  - **(XRP_CLASS_DESCRIP):** Contains the description of the origin user application. This designation is optional.
  - **XRP_CLASS_LABEL:** Contains an IDP which is used to send back the result of the resolve process, e.g. the IDP opening an information channel to the desired target.

Additionally, the encapsulation brick is responsible to receive and forward data messages (payload) after having opened an information channel by a resolve process successfully. This data message is an arbitrary message and obviously has no defined format. While leaving the functional block of IP, it will be encapsulated with an IP header.

- The XRP message received by the checksum brick is depicted in figure 4.6:

  This is the interface between the encapsulation and the checksum computing algorithm.



XRP_CMD_DATA
XRP_CLASS_MESSAGE
XRP_CLASS_LABEL

Figure 4.6: XRP message received by the checksum brick

- The *DATA* message is encoded as follows:
  - **XRP_CMD_DATA:** Identifies a *DATA* message.
  - **XRP_CLASS_MESSAGE:** Contains the IP header from which one wants to compute the checksum. Actually, an arbitrary message with 20 bytes in length could be included.
  - **XRP_CLASS_LABEL:** Contains the IDP used to send back the checksum.

- Figure 4.7 pictures out the XRP messages received by the forwarding table brick:

  This is the interface between encapsulation and the forwarding table as well as between routing and the forwarding table.

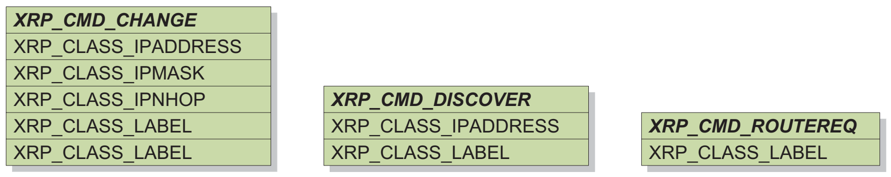


Figure 4.7: XRP messages received by the forwarding table brick

- The *CHANGE* message is encoded as follows:
  - **XRP_CMD_CHANGE:** Identifies a *CHANGE* message.
  - **XRP_CLASS_IPADDRESS:** Contains the subnet address.
  - **XRP_CLASS_IPMASK:** Contains the subnet mask.
  - **XRP_CLASS_IPNHOP:** Contains the IP address of the next hop.
  - **XRP_CLASS_LABEL:** Contains the IDP for the according interface, e.g. Ethernet brick.
  - **XRP_CLASS_LABEL:** Contains another IDP used to send back a confirmation of the updating process.
- The *DISCOVER* message is encoded as follows:
  - **XRP_CMD_DISCOVER:** Identifies a *DISCOVER* message.
  - **XRP_CLASS_IPADDRESS:** Contains the destination IP address of an IP datagram which has to be forwarded.
  - **XRP_CLASS_LABEL:** Contains the IDP used to send back the result of the discover process, e.g. the next hop and the IDP for the according interface, e.g. Ethernet brick.
- The *ROUTEREQ* message is encoded as follows:
  - **XRP_CMD_ROUTEREQ:** Identifies a *ROUTEREQ* message.
  - **XRP_CLASS_LABEL:** Contains the IDP used to send back all entries of the forwarding table.

- The XRP message received by the RIP brick can be seen in figure 4.8:

  These messages characterize no additional interface but describes the routing advertisements for the routing process.

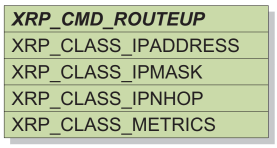


Figure 4.8: XRP messages received by the RIP brick

  – The *ROUTEUP* message is encoded as follows:
    - **XRP_CMD_ROUTEUP:** Identifies a *ROUTEUP* message.
    - **XRP_CLASS_IPADDRESS:** Contains the subnet address.
    - **XRP_CLASS_IPMASK:** Contains the subnet mask.
    - **XRP_CLASS_IPNHOP:** Contains the IP address of the next hop.
    - **XRP_CLASS_METRICS:** Contains the amount of hops to the corresponding subnet through the contained next hop.

    Note that RIP actually exchanges these advertisements between nodes. While leaving the node the *ROUTEUP* message will be encapsulated with an IP header and thus becomes a *data* message because in this Master Thesis RIP is implemented as an IP user application.

4.2.4.2 Between Nodes

- Figure 4.9 shows the IP datagrams of *resolve request* and *resolve response* messages:

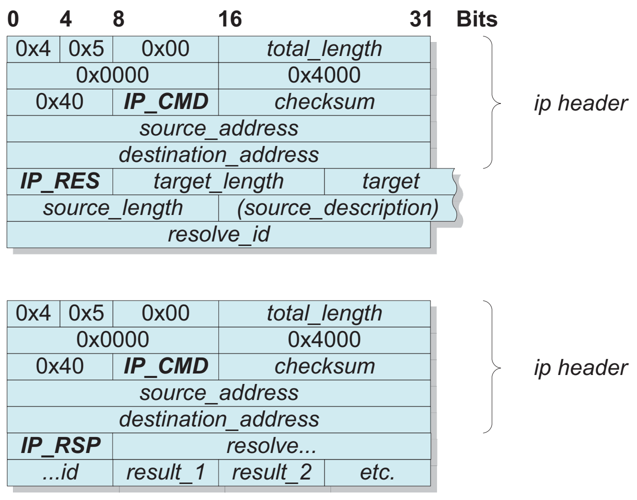


Figure 4.9: IP datagrams of resolve request (top) and resolve response (bottom) messages

Both message types carry of course an IP header in front. The IP header contains a special flag in the next header field as described in the design section 4.1. The IP_CMD code in the next header field identifies this special type of message and leads to a special treatment which differs from *data* messages. In order to distinguish between *resolve request* and *resolve response* messages, one has to look at the first byte of the payload.

- *resolve request* message:
    - **IP_RES:** This flag identifies a *resolve request* message. It is the first byte of the payload.
    - **target_length:** The length of the description of the desired target, declared in bytes.
    - **target:** The description of the desired target.
    - **source_length:** The description length (in bytes again) of the origin user application.
    - **source_description:** The description of the source. This information is optional. If there is no description of the source given, the description length is zero.
    - **resolve_id:** The resolve ID is a random number generated while sending a *resolve request* message and identifies each resolve process in order to deal with further *resolve response* messages.
- *resolve response* message:
    - **IP_RSP:** This flag identifies a *resolve response* message. It is the first byte of the payload.
    - **resolve_id:** The resolve ID of the preceding *resolve request* message.
    - **result_1, ...:** The next header codes of each target found by the resolve process (explained in 4.1).

• The IP datagrams of *data* messages are depicted in figure 4.10:

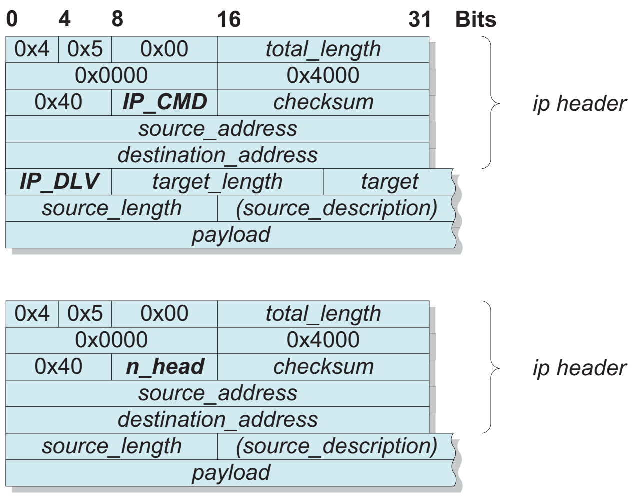


Figure 4.10: IP datagrams of broadcast (top) and unicast (bottom) data messages

This message type can appear in two different modalities. The difference is that a preceding resolve process could have been done in two ways. On the one hand, the channel type of a preceding resolve process could have been 'b' or 'm' in order to open an information channel for multiple targets. On the other hand, the channel type was either 'u' or 'a' in order to reach exactly one target. More on channel types can be found in section 3.3 or 4.1. Hence, for a broadcast *data* message for multiple targets, one can not use the next header field of the IP header but an additional target description is included. For a unicast *data* message for exactly one target one can use the next header field in order to deliver the message.

- *data* message for multiple targets:
  - **IP_DLV:** This flag identifies a *data* message for multiple targets. It is the first byte of the payload.
  - **target_length:** The length of the description of the desired targets, declared in bytes.
  - **target:** The description of the desired targets.
  - **source_length:** The description length (in bytes again) of the origin user application.
  - **source_description:** The description of the source. This information is optional. If there is no description of the source given, the description length is zero.
  - **payload:** The actual payload of the *data* message.
- *data* message for exactly one target:
  - **n_head:** This is the next header code int the next header field of the IP header in order to deliver the *data* message to the desired target.
  - **source_length:** The description length (in bytes) of the origin user application.
  - **source_description:** The description of the source. This information is optional. If there is no description of the source given, the description length is zero.
  - **payload:** The actual payload of the *data* message.

4.2.5 The Bricks

This part deals with each brick separately. It gives a brief description of the purpose of every brick and additionally a well-explained illustration of the internal data processing. If one wants to know more about certain data processing, refer to the comments in the program source code and to the doxygen documentation in the appendix C.

4.2.5.1 Encapsulation Brick

The encapsulation brick is the main brick of the functional block of IP. It deals, as the name implies, with the encapsulation of IP datagrams. Furthermore, it processes resolve requests in order to open information channels to the desired targets. The encapsulation brick is also responsible to deliver messages correctly, e.g. it is concerned about forwarding and local delivery.

To become part of the architecture, there are eight different kind of messages the encapsulation brick is able to receive regularly during lifetime:

- *PUBLISH* messages
- *UNPUBLISH* messages
- *RESOLVE* messages
- Data messages (payload) from a local user application
- *CONFIG* messages
- *resolve request* messages
- *resolve response* messages
- *data* messages from a neighbour node

there exist two different data receive IDPs in order to separate messages from neighbour nodes and from a local brick. The data receive IDP for the neighbour nodes is published in the Ethernet brick and the one for local bricks is published in the KVR of the MINMEX.

Receiving a *PUBLISH* message

As soon as the encapsulation brick receives a *PUBLISH* message from a user application the following steps, depicted in figure 4.11, are performed:

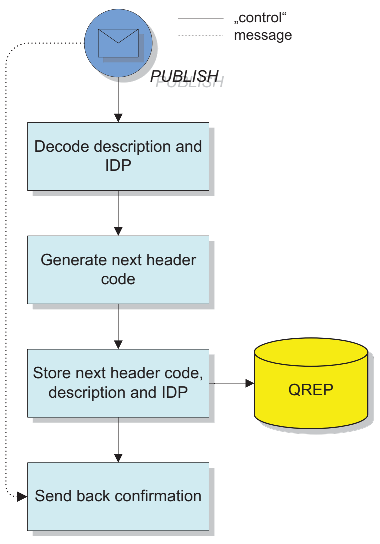

„control"
message
PUBLISH
Decode description and IDP
Generate next header code
Store next header code, description and IDP
QREP
Send back confirmation

Figure 4.11: Data processing of the encapsulation brick receiving a PUBLISH message

1. First of all, the description and according IDP are decoded.
2. A unique next header code for the next header field for the origin user application is generated. This is done by dint of the random number generator and it is corresponding to the next header field in the IP header one byte in size. Additionally, it is made sure that the next header differs from IP_CMD and from the existing next header codes of the user applications already published.
3. The next header code, the description and the according IDP are stored in the QREP of the encapsulation brick in order to be able to deliver further *data* messages to the user application.
4. Finally, the *PUBLISH* message is sent back as a confirmation, if a reply IDP is declared.

Receiving an *UNPUBLISH* message

As soon as the encapsulation brick receives an *UNPUBLISH* message from a user application, the proximate instructions, showed in figure 4.12, are executed:

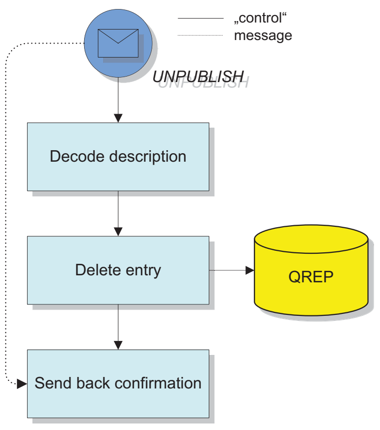


Figure 4.12: Data processing of the encapsulation brick receiving an UNPUBLISH message

1. The description is decoded.
2. The entry is deleted from the QREP.
3. In the end, the *UNPUBLISH* message is sent back as a confirmation if an IDP to use for answering is declared.

Receiving a *RESOLVE* message

Figure 4.13 indicates the details run by receiving a *RESOLVE* message from a user application which intends to open an information channel:

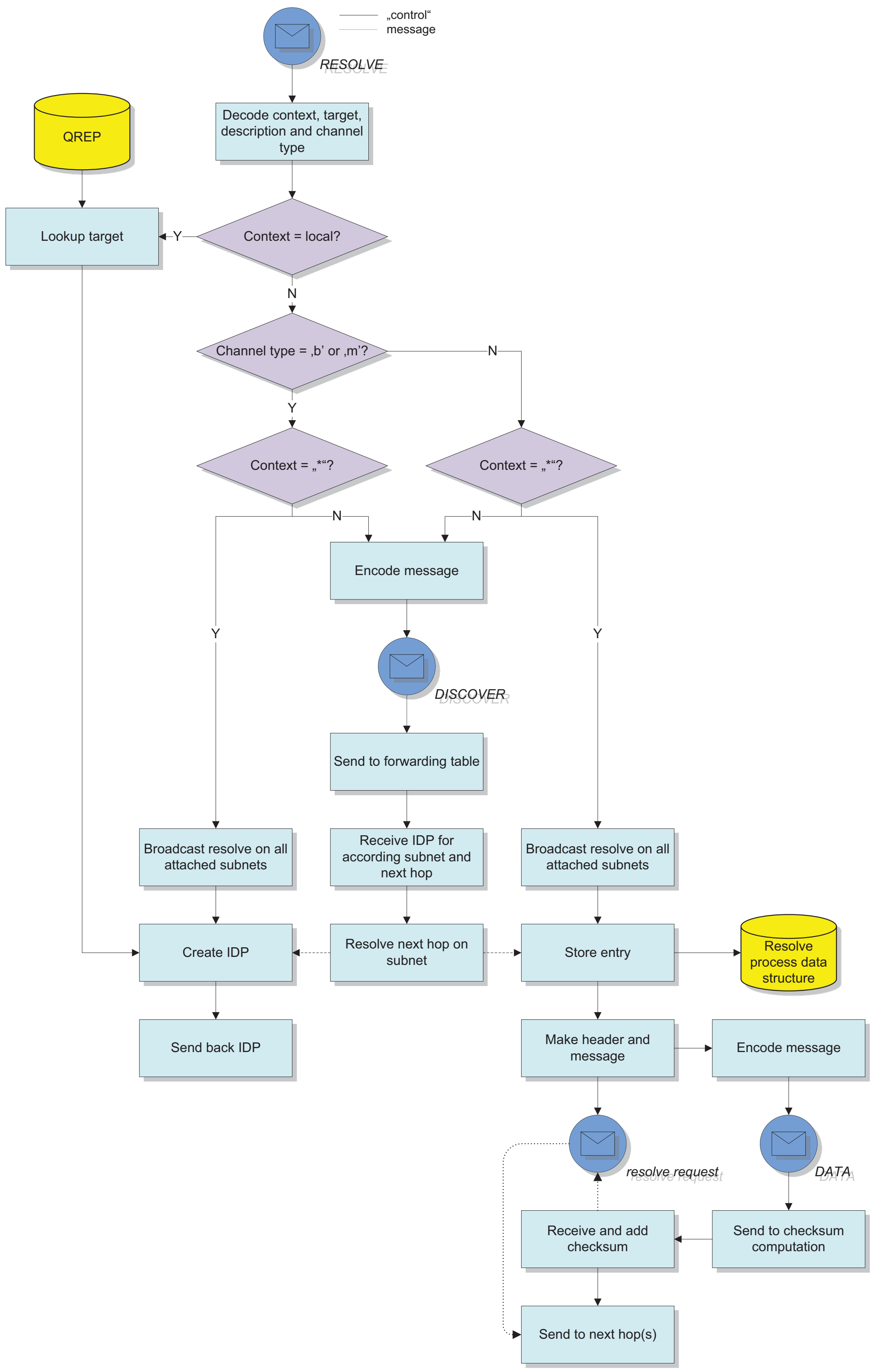

„control“ message
RESOLVE
Decode context, target, description and channel type
QREP
Lookup target
Y
Context = local?
N
Channel type = ‚b‘ or ‚m‘?
N
Y
Context = „*“?
Context = „*“?
N
N
Encode message
Y
Y
DISCOVER
Send to forwarding table
Broadcast resolve on all attached subnets
Receive IDP for according subnet and next hop
Broadcast resolve on all attached subnets
Create IDP
Resolve next hop on subnet
Store entry
Resolve process data structure
Send back IDP
Make header and message
Encode message
resolve request
DATA
Receive and add checksum
Send to checksum computation
Send to next hop(s)

Figure 4.13: Data processing of the encapsulation brick receiving a RESOLVE message

1. If the context is local, the desired target is looked up in the QREP. On success, an IDP for the information channel will be created and sent back to the user application.
2. Otherwise, the context is not local and it depends further on the channel type. If the channel type is 'b' or 'm', the instructions are addicted to the context again:
   (a) When the context is "*", a broadcast resolve on all attached subnets is performed and the resulting IDP for the desired information channel will be sent back to the user application.
   (b) When the context differs from "*", one has to discover a specific node. Therefore, a *DISCOVER* message is encoded and sent to the forwarding table brick. With the response of the forwarding table brick, the next hop can be resolved and the resulting IDP is sent back to the user application.
3. Else, the channel type is 'u' or 'a'. This means that the encapsulation brick has to send a *resolve request* message to make sure that one really receives a *resolve response* message. Depending on the context again:
   (a) When the context is "*", a broadcast resolve on all attached subnets is performed, the resolve process is stored in the resolve process data structure (appendix C) and a *resolve request* message is sent to the next hops.
   (b) When the context differs from "*", a *DISCOVER* message is sent to the forwarding brick in order to discover the next hop. The resolve process is stored in the resolve process data structure and a *resolve request* message is sent to the next hop.

Receiving a data message (payload) from a local user application

As soon as the encapsulation brick receives a data message (payload) from a user application after having successfully opened an information channel, the following steps, depicted in figure 4.14, are performed:

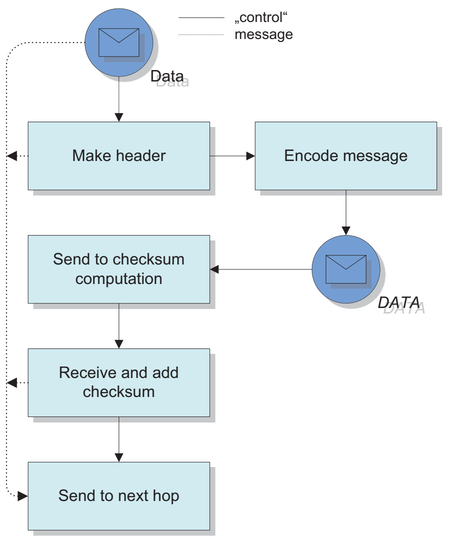


Figure 4.14: Data processing of the encapsulation brick receiving a data message from a local user application

1. Simply, the IP header with the correct checksum is added in front of the payload.
2. Then, the message is sent to the target.

Receiving a *CONFIG* message

When the encapsulation brick receives a *CONFIG* message from the configuration brick, the proximate details, as one can see in figure 4.15, are executed:

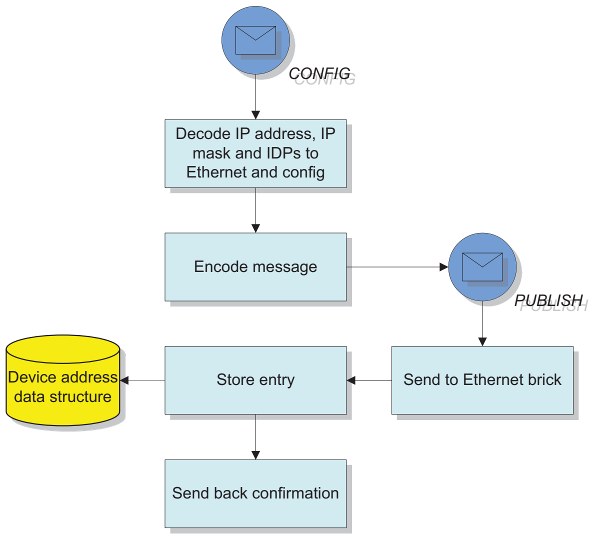

CONFIG
Decode IP address, IP mask and IDPs to Ethernet and config
Encode message
PUBLISH
Device address data structure
Store entry
Send to Ethernet brick
Send back confirmation

Figure 4.15: Data processing of the encapsulation brick receiving a CONFIG message

1. First of all, the IP address, the subnet mask and the IDP of the according interface, e.g. Ethernet brick are, decoded.
2. A *PUBLISH* message is sent to the Ethernet brick containing ”ip” and the IP address in order to be reachable through Ethernet. This step pictures out the building of a dynamic protocol stack.
3. The information are stored in the device address data structure (appendix C). This data structure is needed in order to trace all the attached subnets and to know the configured IP addresses.
4. Finally, a confirmation is sent back.

Receiving a *resolve request* message

As soon as the encapsulation brick receives a *resolve request* message from a neighbour node the following steps, depicted in figure 4.16, are performed:

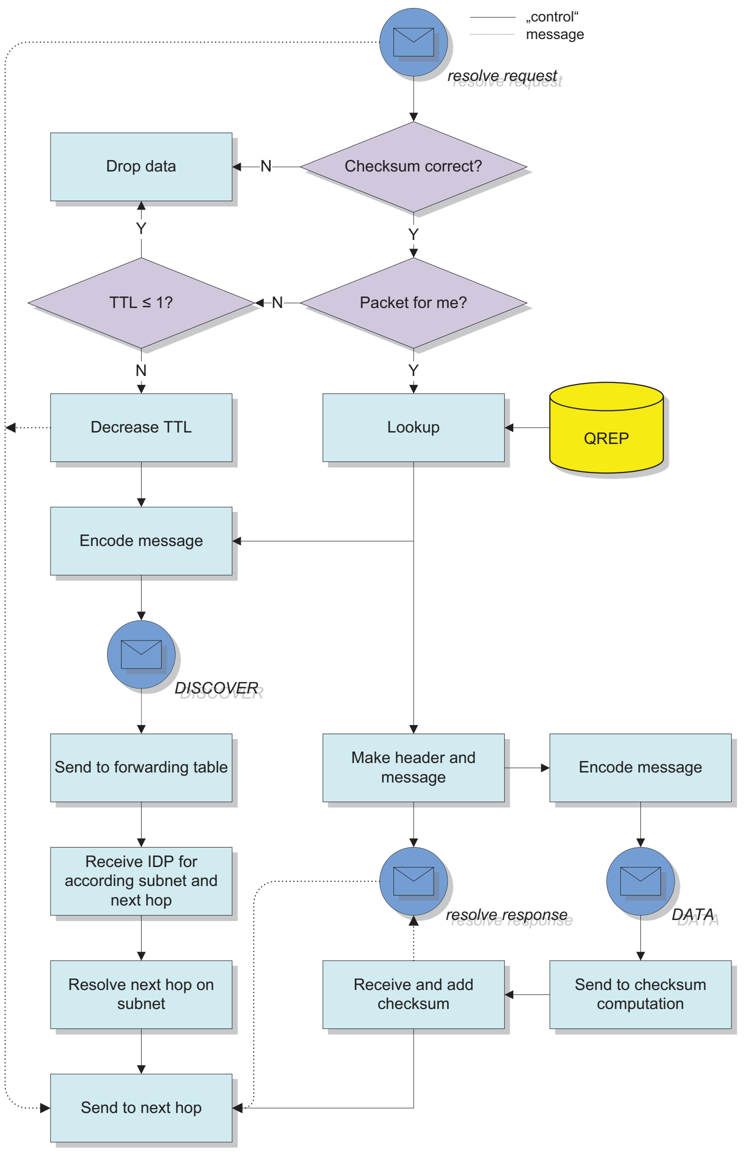


Figure 4.16: Data processing of the encapsulation brick receiving a resolve request message

1. After validating the checksum and the TTL, the encapsulation brick has to decide if the message is for itself or has to be forwarded.
2. If the message has to be forwarded, the following details happen:
   (a) The TTL is decreased by one.
   (b) A *DISCOVER* message is encoded and sent to the forwarding table.
   (c) With the result of the discover process, the next hop for the message can be resolved.
   (d) The message is forwarded to the next hop.
3. Otherwise, the message is meant for this specific node:
   (a) The desired target is looked up in the QREP.
   (b) A *resolve response* message is made containing the resolve ID and the found resolve results. Obviously, a correct checksum is added as well.
   (c) The *resolve response* message is sent back to the origin node.

Receiving a *resolve response* message

Figure 4.17 indicates the instructions executed by receiving a *resolve response* message from a neighbour node for a specific *resolve request* message:

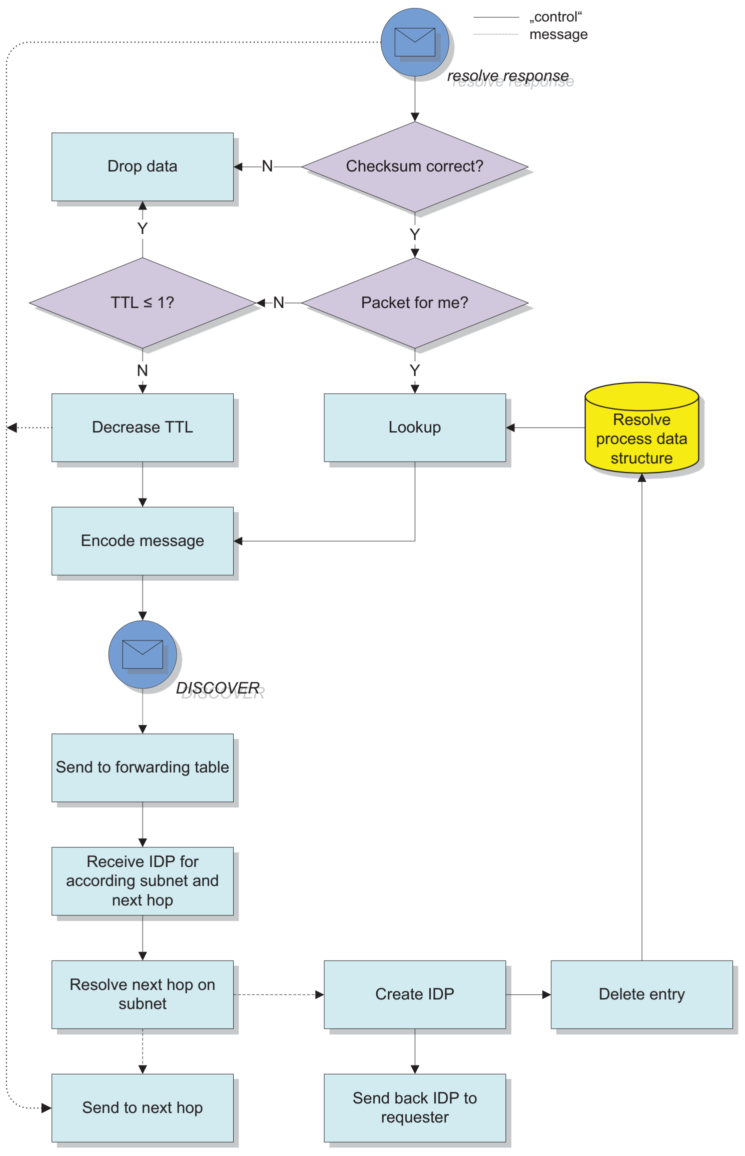


Figure 4.17: Data processing of the encapsulation brick receiving a resolve response message

1. After validating the checksum and the TTL, the encapsulation brick has to decide if the message is for itself or has to be forwarded.
2. If the message needs to be forwarded, the following details are executed:
   (a) The TTL is decreased by one.
   (b) A *DISCOVER* message is encoded and sent to the forwarding table.
   (c) With the result of the discover process, the next hop for the message can be resolved.
   (d) The message is forwarded to the next hop.
3. Otherwise, the message is addressed for this specific node:
   (a) The necessary information are looked up in the resolve process data structure (appendix C).
   (b) A *DISCOVER* message with the IP address of the sender of the *resolve response* message is encoded and sent to the forwarding table brick. With the answer from the forwarding table, one is able to know which interface has to be used in order to communicate with the sender node which contains the resolved target. This is necessary, because the *resolve response* message itself does not contain this mandatory information.
   (c) An IDP which opens an information channel to the desired target is created and sent back to the origin requester of the resolve process.
   (d) Finally, the entry for the resolve process in the resolve process data structure is deleted.

Receiving a *data* message from a neighbour node

As soon as the encapsulation brick receives a *data* message from a neighbour node the following details, depicted in figure 4.18, are performed:

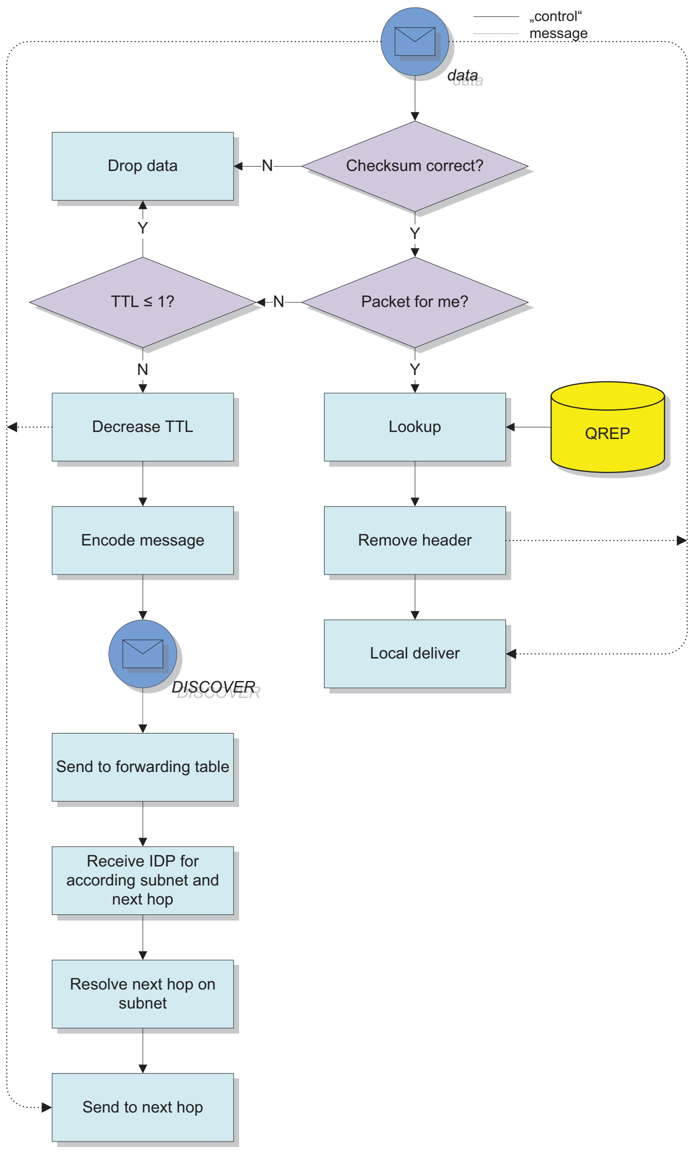


Figure 4.18: Data processing of the encapsulation brick receiving a data message from a neighbour node

1. After validating the checksum and the TTL, the encapsulation brick has to decide if the message is for itself or has to be forwarded.
2. If the message has to be forwarded, the following details are run:
   (a) The TTL is decreased by one.

(b) A *DISCOVER* message is encoded and sent to the forwarding table.

(c) With the result of the discover process, the next hop for the message can be resolved.

(d) The message is forwarded to the next hop.

3. Otherwise, the message is addressed for this specific node:

(a) The target given (either by the next header code given in unicast *data* messages or the description given in broadcast *data* messages) is looked up in the QREP.

(b) The IP header is removed from the message.

(c) The resulting payload message is delivered locally to the target user application.

4.2.5.2 Checksum Brick

The checksum brick is responsible for checksum computation. It uses the Internet checksum algorithm [17] and sums up blocks of two bytes per message. Even though it anticipates an IP header, an arbitrary message of 20 bytes in length can be sent to the brick.

To become part of the architecture, there is only one kind of message the checksum brick is able to receive during lifetime:

- *DATA* messages

Receiving a *DATA* message

As soon as the checksum brick receives a *DATA* message, the proximate instructions, depicted in figure 4.19, are performed:

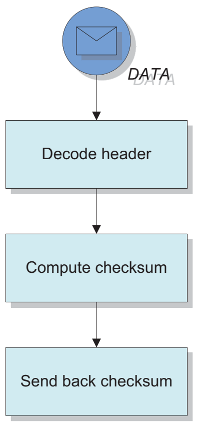


Figure 4.19: Data processing of the checksum brick receiving a DATA message

1. The message is extracted from the XRP framing.
2. The checksum is computed.
3. And finally, the checksum is sent back to the reply IDP.

4.2.5.3 Forwarding Table Brick

The forwarding table brick holds, as the name implies, the forwarding table (implemented as QREP). It is used to help the encapsulation brick forwarding IP datagrams to the accurate next hop. Therefore, the forwarding brick is responsible for the correct forwarding process.

To become part of the architecture, there are three different kind of messages the forwarding table brick is able to receive during lifetime:

- *CHANGE* messages
- *DISCOVER* messages
- *ROUTEREQ* messages

Each message is received at the same data receive IDP published in the KVR of the MINMEX.

Receiving a *CHANGE* message

Figure 4.20 shows the instructions executed by receiving a *CHANGE* message from a possible routing process, e.g. the RIP brick:

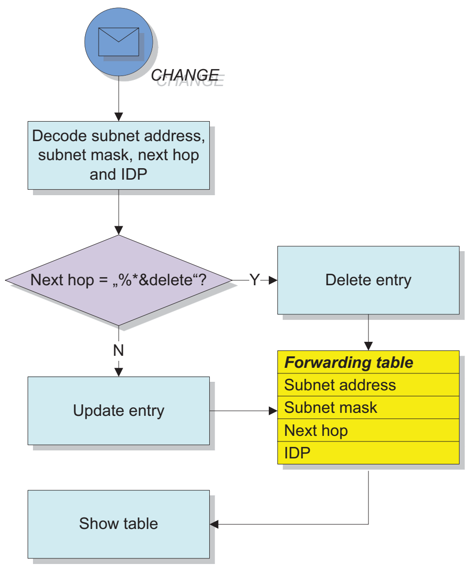


Figure 4.20: Data processing of the forwarding table brick receiving a CHANGE message

1. First of all, the subnet address, the subnet mask, next hop and according IDP for the interface, e.g. the Ethernet brick, are decoded.
2. When the next hop equals "%*&delete", the entry for this subnet will be deleted. In this case, the entry suffers from missing route updates, hence the route died. "%*&delete" acts as a magic string.
3. Otherwise, the entry for the certain subnet is updated and the forwarding table is printed on the brick output.

Receiving a *DISCOVER* message

When the forwarding table brick receives a *DISCOVER* message, the following details, as one can see in figure 4.21, are run:

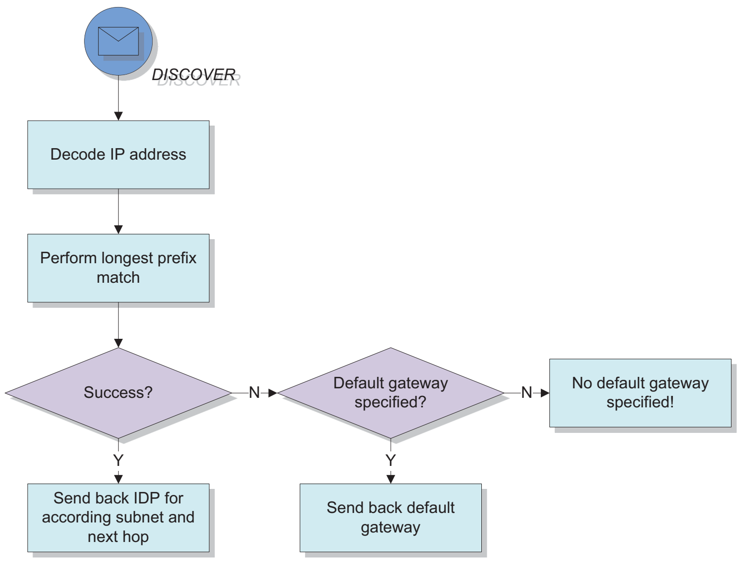

DISCOVER
Decode IP address
Perform longest prefix match
Success?
N
Default gateway specified?
N
No default gateway specified!
Y
Send back IDP for according subnet and next hop
Y
Send back default gateway

Figure 4.21: Data processing of the forwarding table brick receiving a DISCOVER message

1. Foremost, the IP address is decoded.
2. A longest prefix match is performed.
3. If this ends up in a success, the discovered next hop and the IDP for the according interface, e.g. Ethernet brick, is sent back.
4. Otherwise, the default gateway is sent back, if one is specified. If no default gateway is specified, a timeout occurs.

Receiving a *ROUTEREQ* message

As soon as the forwarding table brick receives a *ROUTEREQ* message, the following instructions, illustrated in figure 4.22, are performed:

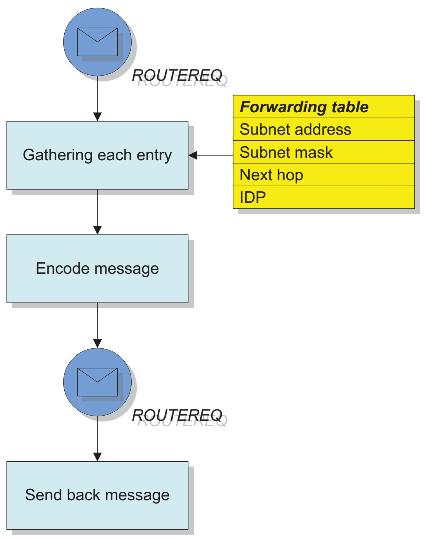


Figure 4.22: Data processing of the forwarding table brick receiving a ROUTEREQ message

1. The forwarding table brick gathers every entry of the forwarding table.
2. The result is encoded as an XRP message and sent back to the requester, e.g. the RIP brick.

4.2.5.4 RIP Brick

The RIP brick is concerned about routing issues. As the name implies, it implements RIP in ANA. Moreover, the RIP brick updates the forwarding table continuously. The route updates are triggered upon a change in the routing table as well as sent by a periodic timer (implemented with ANA timers).

To become part of the architecture, there is just one kind of message the RIP brick receives regularly during lifetime:

- *ROUTEUP* messages

Each message is received at the same data receive IDP published in the KVR of the MINMEX.

Receiving a *ROUTEUP* message

Figure 4.23 indicates the instructions executed by receiving a *ROUTEUP* message:

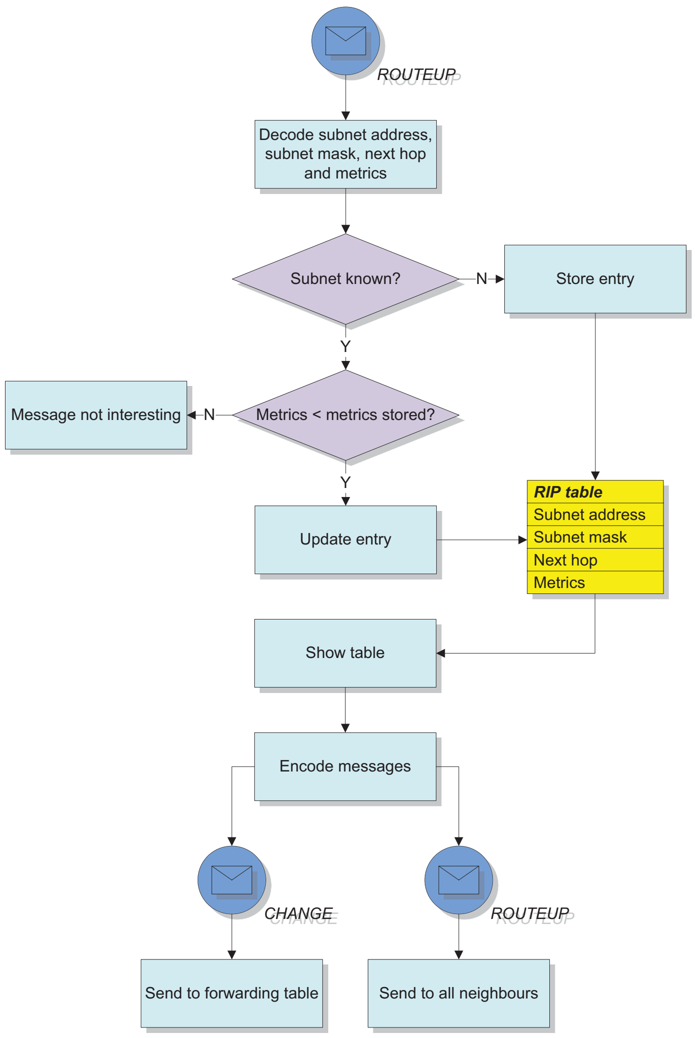


Figure 4.23: Data processing of the RIP brick receiving a ROUTEUP message

1. The subnet address, the subnet mask, the next hop and the metrics are decoded.
2. If the subnet is unknown, the entry is stored in the routing table.
3. Else, the subnet is known and the entry of this specific subnet is updated providing the metrics is lower than the ones stored. Should the metrics not be interesting, the message will be dropped.
4. When the *ROUTEUP* message leads to a change in the routing table, the table is shown and two types of XRP messages are encoded and sent:
   (a) A *ROUTEUP* message in order to dispread the new routing information. Thus, this message will be encapsulated with an IP header and will leave the node.

(b) A *CHANGE* message for the local forwarding table brick in order to update the forwarding table.

Besides the instructions triggered by an event as described above, the brick owns a timer function in order to run periodic route updates for the neighbour nodes. Moreover, there is also a garbage collecting function existing which tidies up with route entries, suffering from missing updates.

4.2.5.5 Sample User Brick

This Master Thesis uses a sample user brick to be able to implement and also validate the architecture. The sample user brick represents a specific user application of the functional block of IP. To sum it up, the sample user brick does the following:

- Publishes "ip_usr" along with a receive data IDP in the KVR of the MINMEX.
- Sends a message to another entity of the sample user brick on an arbitrary node:
  - Sends a *RESOLVE* message to the encapsulation brick with a specific context, "ip_usr" as target, channel type 'u' and the source description "ip_usr".
  - Upon receiving back an IDP to the desired target, it sends a data message containing a string.
- Displays all the received messages on the receive data IDP along with the IP address of the sender and the description.

One is able to choose the specific context for the resolve request at runtime by typing an auxiliary command line argument:

```
$ ./bin/ip_usr -n MINMEX_CONTROL_GATE -a CONTEXT
```

4.3 Validation

This section demonstrates the validation of the implementation of IP and RIP in ANA. In order to validate the implementation, this Master Thesis used the TIK testbed[1] to install an ANA network running IP and RIP. To configure an arbitrary topology the vlink brick [4] for setting up virtual links has been used. One can find more instructions how to run IP and RIP in ANA in the appendix A. The resulting ANA network and IP compartment has been taken to run several deciding scenarios.

- First of all, this Master Thesis used the sample user brick described before in order to send simple string messages from one node to another in the same IP subnet. The resulting IP datagrams all looked accurate and didn't face any problems in finding the targets immaculate.
- Moreover, the same test has been accomplished with two subnets as a first interesting test, highlighting the novel feature for an ANA network. Therefore, the messages have been sent over two hops, with a router in between. An ANA IP router only needs to start IP and RIP in ANA but no user application is necessary. All IP datagrams found their way to the desired targets over one hop properly.
- Furthermore, sending IP datagrams over just two hops is not enough. Now, the messages are sent over four hops. Thus, a topology with five nodes (including three routers) has been set up. The IP datagrams all looked good again and followed their way to the desired targets.

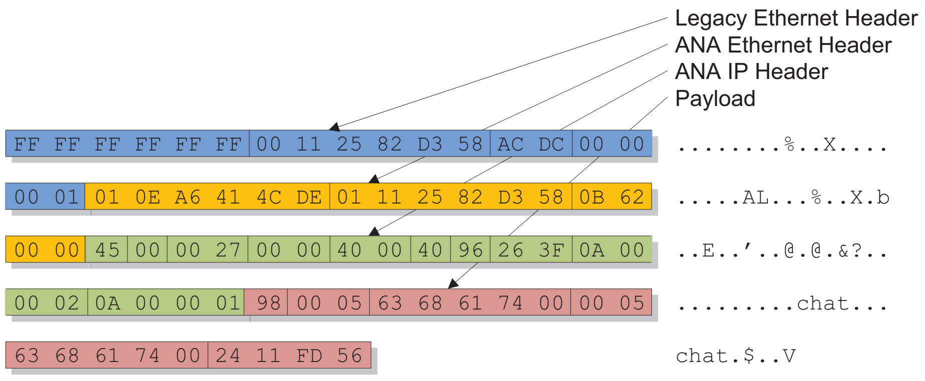
[1] A collection of hosts attached to both wired Ethernet and wireless LAN interfaces [39]

- Additionally, this Master Thesis intended to validate an application scenario. Hence, a chat application [2] was used this time in order to generate IP traffic. The ANA network consisted of three nodes in three different subnets as well as a router connecting them. Furthermore, this validation scenario was tested with wired Ethernet and wireless LAN interfaces. In the wired scenario all IP datagrams found their way to the desired targets via router properly, whereas in the wireless scenario, the IP datagrams encountered difficulties. The packet loss in wireless networks is much higher, therefore IP datagrams are lost in this validation scenario in some cases because no transport control has been implemented so far.

In the second part of this validation section, this Master Thesis would like to reveal the communication in detail and look at the IP datagrams with a magnifying glass. Therefore, an ANA network and IP compartment is set up with two nodes communicating together by the chat application over IP again. The IP addresses of the nodes are configured as ”10.0.0.1” and ”10.0.0.2”, respectively. The IP datagrams concerning the communication of the nodes were grabbed by wireshark (ethereal) [44]. Now, if the chat application on ”10.0.0.2” wants to send ”Hi there!” to ”10.0.0.1”, one can observe three IP datagrams:

- A *resolve reques*t message from 10.0.0.2 to 10.0.0.1
- A *resolve response* message from 10.0.0.1 to 10.0.0.2
- A *data message* containing ”Hi there!” from 10.0.0.2 to 10.0.0.1

Remember, before one is able to communicate with a desired target, an information channel has to be opened by a resolve process. The IP datagram captured on the wire of the *resolve request* message is depicted in figure 4.24:



```
FF FF FF FF FF FF 00 11 25 82 D3 58 AC DC 00 00    ........%..X....
00 01 01 0E A6 41 4C DE 01 11 25 82 D3 58 0B 62    .....AL...%..X.b
00 00 45 00 00 27 00 00 40 00 40 96 26 3F 0A 00    ..E..'..@.@.&?..
00 02 0A 00 00 01 98 00 05 63 68 61 74 00 00 05    .........chat...
63 68 61 74 00 24 11 FD 56                         chat.$..V
```

Figure 4.24: IP datagram of the resolve request message

One can see the legacy Ethernet header, the ANA Ethernet header, the ANA IP header and the payload of the message. The legacy Ethernet header shows the destination and source MAC address as well as the Ethernet type field:

- The destination MAC address `0xFFFFFFFFFFFF` is the broadcast address. One can notice that all messages for ANA are broadcasted on the wire.
- The source MAC address `0x00112582D358` is the MAC address of the node configured with the IP address 10.0.0.2.

- The Ethernet type field shows `0xACDC`. This is a detection flag in order to highlight communication for ANA (in comparison to `0x0800` which would declare a standard IP datagram or `0x086DD` would declare a IPv6 datagram).

The ANA Ethernet header that shows the destination and source MAC address again as well as the Ethernet type field:

- The destination MAC address `0x010EA6414CDE` is the MAC address chosen by the vlink brick of the node configured with the IP address 10.0.0.1. Note that the first byte is different from the real MAC address because this byte indicates the vlink ID.
- The source MAC address `0x01112582D358` is the MAC address chosen by the vlink brick of the node configured with the IP address 10.0.0.2. Note that the first byte is also different from the real MAC address because this byte indicates the vlink ID again.
- The Ethernet type field shows `0x0B62`. This is a next header code in order to highlight communication for the IP compartment in ANA.

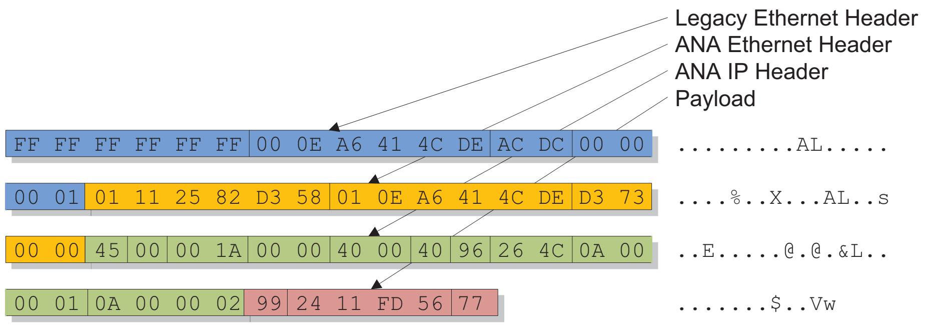
The ANA IP header looks as follows:

- `0x45` indicates the version set to ”4” and the header length set to ”5”.
- The ToS field is set to zero.
- The total length shows `0x0027` which one can convert to 39 bytes. Counting them all shows that this value is correct.
- The field concerning fragmentation is `0x00004000` corresponding to a ”Do not fragment” flag.
- The TTL shows `0x40` which one can convert to ”64”. This value is also correct.
- The next header code shows `0x96` which declares an IP_CMD message what is correct as well.
- Then, the checksum `0x263F`, the source and destination IP addresses are shown which all are correct.

Finally the payload of this *resolve request* message looks as follows:

- The first byte contains `0x98` which identifies a valid *resolve request* message.
- Then, the length of the target description (`0x0005`) and the target description itself (`0x6368617400` = ”chat”) are present.
- Not only the length of the source description but also the source description itself look exactly the same.
- Finally, `0x2411FD56` stands for the resolve ID for this resolve process.

The IP datagram captured on the wire of the *resolve response* message is pictured in figure 4.25:



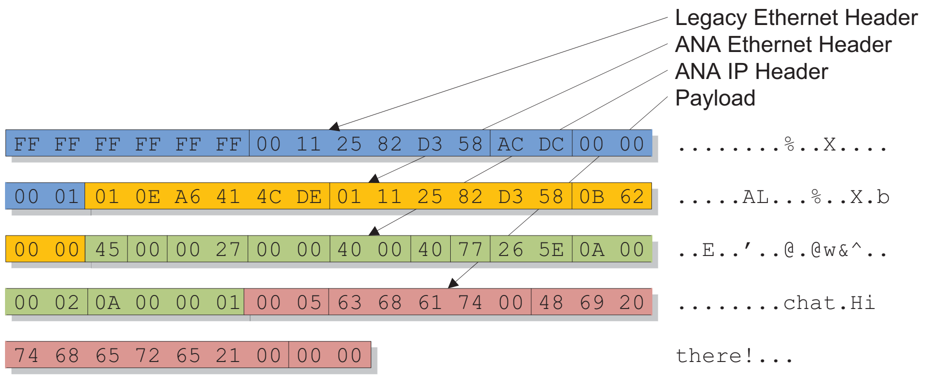
```
FF FF FF FF FF FF 00 0E A6 41 4C DE AC DC 00 00   .........AL.....
00 01 01 11 25 82 D3 58 01 0E A6 41 4C DE D3 73   ....%..X...AL..s
00 00 45 00 00 1A 00 00 40 00 40 96 26 4C 0A 00   ..E.....@.@.&L..
00 01 0A 00 00 02 99 24 11 FD 56 77               .......$..Vw
```

Figure 4.25: IP datagram of the resolve response message

Again, one can observe the legacy Ethernet header, the ANA Ethernet header, the ANA IP header and the payload of the message. It is not necessary to explain everything again, except for the payload:

- The first byte contains `0x99` which identifies a *resolve response* message and is correct.
- Then, the resolve ID `0x2411FD56` is given. Note that it is naturally identical to the one in the *resolve request* message.
- And finally, the discovered next header code is `0x77`.

Now the wanted information channel from 10.0.0.2 to the desired target on 10.0.0.1 is opened and the actual communication can start. The figure 4.26 shows the IP datagram of the *data* message:



```
FF FF FF FF FF FF 00 11 25 82 D3 58 AC DC 00 00   ........%..X....
00 01 01 0E A6 41 4C DE 01 11 25 82 D3 58 0B 62   .....AL...%..X.b
00 00 45 00 00 27 00 00 40 00 40 77 26 5E 0A 00   ..E..'..@.@w&^..
00 02 0A 00 00 01 00 05 63 68 61 74 00 48 69 20   ........chat.Hi
74 68 65 72 65 21 00 00 00                        there!...
```

Figure 4.26: IP datagram of the data message

Interesting to explain:

- The next header code in the next header field of the IP header shows `0x77` as expected.
- The message sent was `0x48692074686572652100` = ”Hi there!”.

The validation is a resounding success and the implementation works accurately.

4.4 Summary

IP as the first use case for a complex network protocol, extending the features of ANA increasingly, was successfully designed and implemented in the ANA Playground. Because of the IP compartment it is now possible to send messages over disparate subnets. Furthermore, RIP provides the leadoff opportunity to perform routing in an ANA network.

IP and RIP in ANA exhibits some variances compared to the standard IP. It does not offer the full functionality of the standard IP. Nevertheless, the most important features like the IP header, encapsulation, checksum computation, forwarding and the addressing scheme are supported. Additionally, RIP offers the full functionality of the standard RIP.

Thanks to the modular concept, the decomposition discloses exceeding flexibility whereas each protocol piece can be exchanged and extended effortlessly. The architecture of one node participating in the compartment contains four individual and specialized bricks, separating and modeling the fundamental functionality of the protocol:

- Encapsulation brick: The encapsulation brick is the main brick of the functional block and handles the IP headers and the encapsulation of the IP datagrams. Moreover, it is responsible to deliver data messages locally and to forward them to remote nodes.
- Checksum brick: The checksum brick computes the checksums.
- Forwarding table brick: The forwarding table brick holds the forwarding table and is therefore responsible for the accurate forwarding of the forwarding process.
- RIP brick: The RIP brick is executing routing in ANA and updates the forwarding table continuously.

Together, the explained bricks build the functional block of IP and RIP. The communication, hence the interfaces between the bricks and the nodes, are elaborately defined in order to provide the aspired flexibility. Node internal communication is encoded as XRP messages with well-defined commands and classes whereas node external communication is achieved by genuine IP datagrams.

In the end, several crucial validation scenarios exemplify the respectable implementation of IP and RIP in ANA. Forwarding paths as well as the format of IP datagrams are validated correctly by this Master Thesis.

Chapter 5

Field-Based Service Discovery for ANA

Whereas the first part of this Master Thesis is concerned about the Internet Protocol (IP) in the ANA world, the second part deals with an exploring novel protocol type not yet implemented in today's Internet. Field-based service discovery is an intuitive combination of field-based routing explained in 2.5 and publish-subscribe service discovery introduced in 2.4. The ultimate goal is to publish a specific service type network-wide in order to provide address-agnostic service discovery.

The first section of this chapter deals with the design of such a field-based service discovery compartment for ANA. It gives an overview and describes the requirements and processes to achieve the aimed goal as well as the decomposition and predestined fitting in the modular and flexible ANA world. Additionally, it explains the terminology of dynamic containers acting as mandatory communication interfaces. Section 5.2 specifies the precise implementation and development of every brick involved. The field-based service discovery compartment is finally validated in section 5.3 and summarized in the last section of this chapter.

5.1 Design

This section pictures out the design of a field-based service discovery protocol. Besides the description of the protocol itself, it highlights above all the exchangeability of the individual protocol pieces thanks to the modular and flexible decomposition.

5.1.1 Protocol Overview

The ultimate goal of the field-based service discovery compartment is the opportunity of publishing a certain service type network-wide. Thus, client applications are able to discover this specific service type by sending an address-agnostic subscribe message. Moreover, in case of multiple service instances of the same type, the client is guided automatically to the best service instance regarding the network conditions and capacities of service, respectively.

The network-wide publishing is obtained by a scalar field overlay for each service type. A scalar field is analogous to a potential in electrostatics, resulting from electrical point charges. Hence, every node participating in the network computes its potential, respecting a specific service type with a specific point charge considered as the capacity of service (CoS). One is free to choose the potential function building the field. Possibilities are for example an electrostatic field, a magnetic field, a heat field or many other conceivable scalar fields and combinations. The defined distribution holds its maximum at the service instance itself and is monotone sloping along the other nodes building the network.

Naturally, the described field is not static during network lifetime. As soon as multiple service instances of a specific service type are published, the resulting field is a superposition of the individual field quotients. Furthermore, a service instance is able to instantly update the published CoS. Thus, a specific field for a specific service type depends persistently on emergence, disappearance and CoS of every service instance. Moreover, different service types in a network result in multiple field overlays for the network which exist simultaneously and may differ regarding their potential functions.

Routing in the field-based service discovery network is done along the steepest gradient of the field. In order to achieve this, each node gathers all service type potentials from the neighbour nodes. Now, if a client application tries to discover a specific service type, the subscribe message will be forwarded to the next node with the highest potential, until it reaches a service instance. The node containing the service instance holds per definition an infinite potential, therefore the forwarding process will be stopped and a response can be sent back. There are several options for sending back a response, depending on the implementation:

- If the subscribe message carries an identifying address of the client, e.g. an IP or MAC address, the response message can be sent back by means of a different network protocol such as IP or a data link protocol such as Ethernet, when acting inside an Ethernet segment. The response message may carry the according address of the service instance as well, so further communication between client and service instance is done over the mentioned accessory protocol. In this case, field-based service discovery is only used for nonrecurring service discovery needs. Thus, field-based service discovery is acting as a pure routing protocol from a client towards a service instance. Field-based routing also can be considered as content-based routing because the routing process along different field overlays is specific to the service type a packet intends to reach.

- However, field-based service discovery is more than this. Another possibility is to store the path from client to service instance in the nodes. Each node perceives from where it receives a subscribe message and forwards the response message back to the same node. As a result, the response will take exactly the same path back to the origin (source routing). A drawback of this case is apparently the situation of a node failure. If one node of the feedback channel is lost, the whole message will be lost.

- A third way of sending back a response is that the client application itself generates an additional field overlay for the network. In this case the response message can be forwarded through the network analogously to the subscribe message. Interestingly, this way defines an entirely novel namespace, resulting in field-based service discovery as full network and routing protocol.

As seen before, field-based service discovery comprehend a lot of different processes which have to be strongly separated for the design of the architecture. One has to distinguish between field assembly (publishing), routing, forwarding and information dissemination between nodes. Every process is independent from the other.

Field assembly is done as soon as a service type is published and every time a new service instance appears, disappears or updates its CoS. It describes the process where each node computes its potential for all the service types. On the other hand, routing describes the process of gathering all the potentials from the neighbour nodes and picking out the one with the highest potential. Forwarding describes the process of delivering payload between nodes by dint of the results of the routing process. Finally, dissemination specifies the way how information is exchanged between nodes. Information in this case is protocol information like field assembly and route update messages.

5.1.2 Architecture and Decomposition

Field-based service discovery contains many different processes. One has to strongly separate between field assembly, routing, forwarding and information dissemination. Therefore, the design and decomposition of such a protocol results in a very modular and flexible architecture. Every piece is independent from each other and is individually exchangeable.

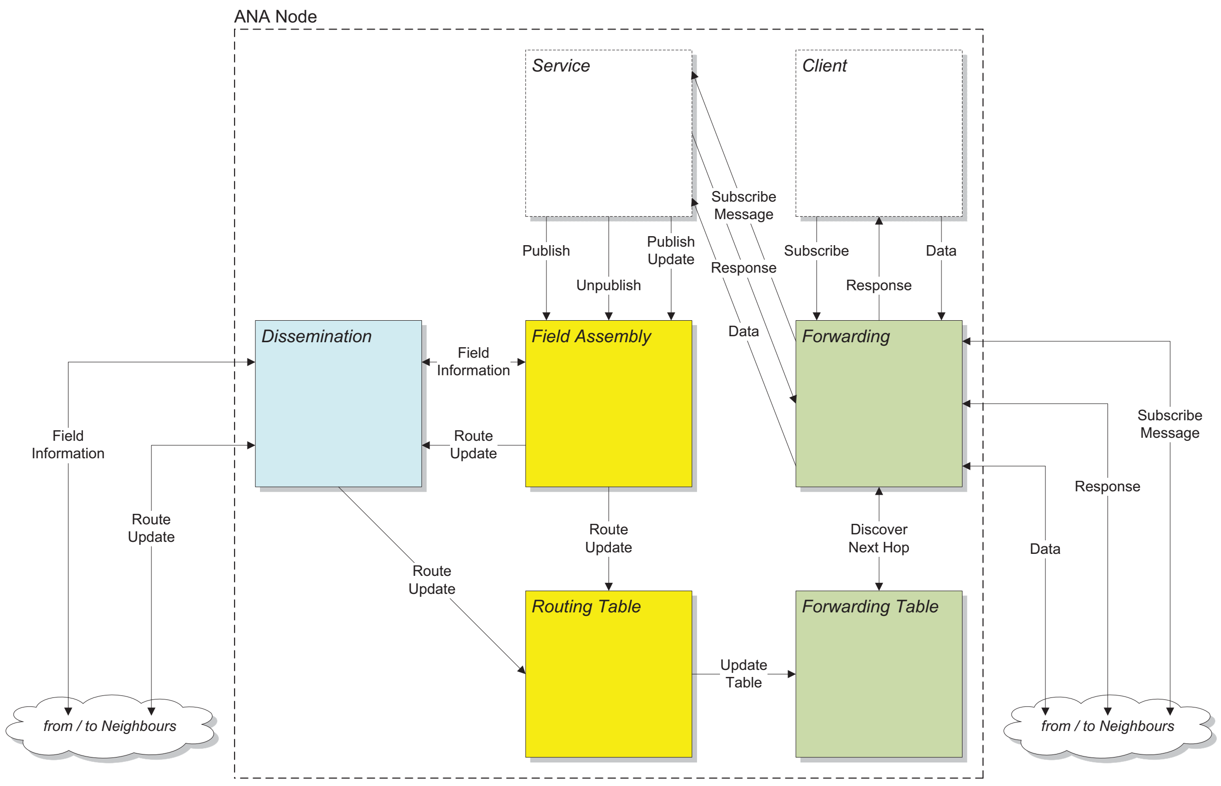


Figure 5.1: Node overview of the field-based service discovery architecture

Depicted in figure 5.1, this Master Thesis strongly distinguishes between five different processes:

- The field assembly process handles the publishing of service instances and thus generates the field assembly information. Additionally, it computes all service type potentials for this specific node by receiving field information from remote service instances.
- The routing table is responsible to know all potentials of the neighbour nodes in order to pick out the one with the highest potential.
- The dissemination process deals with the distribution of information between nodes. Field assembly information and routing information is thereby strongly separated.
- The forwarding process is responsible to forward subscribe messages from client applications as well as payload. Furthermore, it provides a feedback channel for response messages from service instances.

- The forwarding table holds for each service type the next hop with the highest potential because forwarding is done along the steepest gradient of the field. Therefore, the forwarding table supports the forwarding process.

The benefits are obvious and eminent. If one wants to improve the dissemination process, only the dissemination piece of the architecture needs to be exchanged. If one needs to establish a different field type overlay for the network, only the field assembly piece has to be exchanged. If one is not happy with the way the response messages are sent back, only the forwarding piece of the architecture needs to be exchanged. Furthermore, routing and forwarding tables may be used by other protocols and applications if necessary and the whole architecture can be extended easily.

There is also a disadvantage of the architecture existing: If one big process is divided into smaller processes, the overhead communication between the smaller processes is increasing. Nevertheless, there are possibilities to reduce overhead (as seen for example in the implementation of the dissemination brick) but reducing overhead is not a goal of the ANA project indeed.

For the purpose of providing the described flexibility, well-defined communication interfaces between the individual processes are needed. The next part explains the messages between the processes and between nodes in order to make up the functionality of field-based service discovery whereas the precise message formats can be found in the implementation section 5.2.

5.1.3 Communication Interfaces

The described architecture needs specific message exchange between the individual processes in order to function properly. This part describes all the messages existing by design and shown in figure 5.1 whereas the implementation section 5.2 deals with the accurate implementation of the interfaces.

The messages serve as dynamic containers so as to offer a flexible and generic formatting. The formats of the dynamic containers look identical for different variations of the field-based service discovery compartment and information exchange whereas the content can be dynamically chosen regarding the compartment requirements. The unique message names introduced in this part are used throughout the rest of this documentation.

- Publishing and unpublishing a service instance as well as publish updates:

  This is the interface between a service instance and the field-based service discovery compartment.

  - **PUBLISH:** Publish messages are sent by service instances which want to publish itself network-wide in order to give client applications the opportunity to discover the service types.
  - **UNPUBLISH:** Unpublish messages are the opposite of publish messages. A service instance is able to retract the previous publishing.
  - **PUBLISHUP:** Publish updates are necessary for service instances to keep the field overlay alive. Furthermore, they give the chance to update the published CoS.

- Field assembly and route update messages for a service type:

  These are the field and routing information which has to be exchanged between neighbour nodes. Refer to the implementation section 5.2 in order to see how the messages look like inside a node and between nodes.

  - **FIELD:** Field assembly messages are the result of publishing a service instance. They dispread the field information over the network. Thus, the network participants are able to compute their potentials. Field messages are processed by the field assembly process and distributed by the dissemination process.
  - **ROUTEUP:** Route updates contain routing information. As opposed to the field assembly process, the routing process wants to get to know the potentials of all neighbour nodes regarding a specific service type. They are generated by the field assembly process, stored in the routing table and distributed by the dissemination process.

- Subscribe, response and data messages:

  This is the interface between client applications and the field-based service discovery compartment. The messages are payload messages which has to be exchanged between neighbour nodes and are supposed to reach service instances as well as client applications. Refer to the implementation section 5.2 in order to see how the messages look like inside a node and between nodes.

  - **SUBSCRIBE:** A subscribe message is sent by a client application which wants to discover a specific service type. It is forwarded by the forwarding process by dint of the forwarding table.
  - **RESPONSE:** After having successfully discovered a service instance, a response message is sent back to the client application.
  - **DATA:** Now, there is an information channel existing between the client application and the service instance. The client application has discovered a path to the service instance and is able to send data messages through this path.

- Updating the forwarding table and discover its entries:

  This is the interface to the forwarding table which is used by the routing process as well as the forwarding process.

  - **CHANGE:** The change message is sent by the routing table and updates the forwarding table with the routes according to the steepest gradient of the field.
  - **DISCOVER:** In order to discover a route for a specific service type, the forwarding process sends a discover message to the forwarding table.

5.2 Implementation

This section describes the concrete programming of the field-based service discovery architecture in the ANA Playground. It shows which tools have been used and pictures out the development of each brick involved building the functional block of field-based service discovery.

5.2.1 Overview

The decomposition of field-based service discovery fits exactly into the ANA architecture because it is designed with the MINMEX and Playground in a modular and flexible way as well. Field-based service discovery is implemented in the ANA Playground as exploring novel and cutting-edge protocol. The protocol is represented by a functional block. Each process separated and described in the design section is implemented as an individual brick whereas all bricks together form the mentioned functional block. The communication between the bricks as well as the nodes, e.g. the interfaces of the dynamic containers, are encoded as XRP messages, introduced in 3.4.

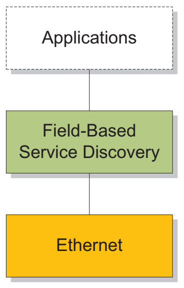


Figure 5.2: Protocol stack of field-based service discovery running on top of Ethernet

Importantly, the field-based service discovery compartment needs another compartment in order to send data packets to the wire and to identify the neighbour nodes. For now, the Ethernet compartment and thus the Ethernet brick [4] is used, as depicted in figure 5.2. Therefore, the dissemination brick (fbr_diss.c) and the forwarding brick (fbr_forw.c), both the bricks which are actually communicating with neighbour nodes, are published in the Ethernet brick. This process describes actually the building of a protocol stack dynamically. The neighbour nodes are found by an AL2 broadcast resolve, explained in section 3.3. Naturally, there are also other data link compartments or even network compartments imaginable, for example the IP compartment. More on this one can find in the chapter 6.2 about further work.

The ANA API functions, explained in section 3.3, supported by the field-based service discovery compartment are publish and unpublish functions:

```
anaLabel_t anaL2_publish(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *description, int descriptionLength,
AL2Callback_t service, int separateThread);

int anaL2_unpublish(const anaLabel_t COMPARTMENT_LABEL,
void *context, int contextLength, void *description, int descriptionLength);
```

Here, publish results in reachability through a certain compartment and compartment-wide publicity similar to the publish in an IP compartment (4.2) but address-agnostic. Furthermore, one can imagine the *subscribe* message as additional API function resulting in opening an information channel to a specific target which is similar to the resolve function.

The figure 5.3 shows a similar illustration as in the design section 5.1. This time, the individual processes are substituted by the different bricks and the communication between the bricks shows the real XRP commands.

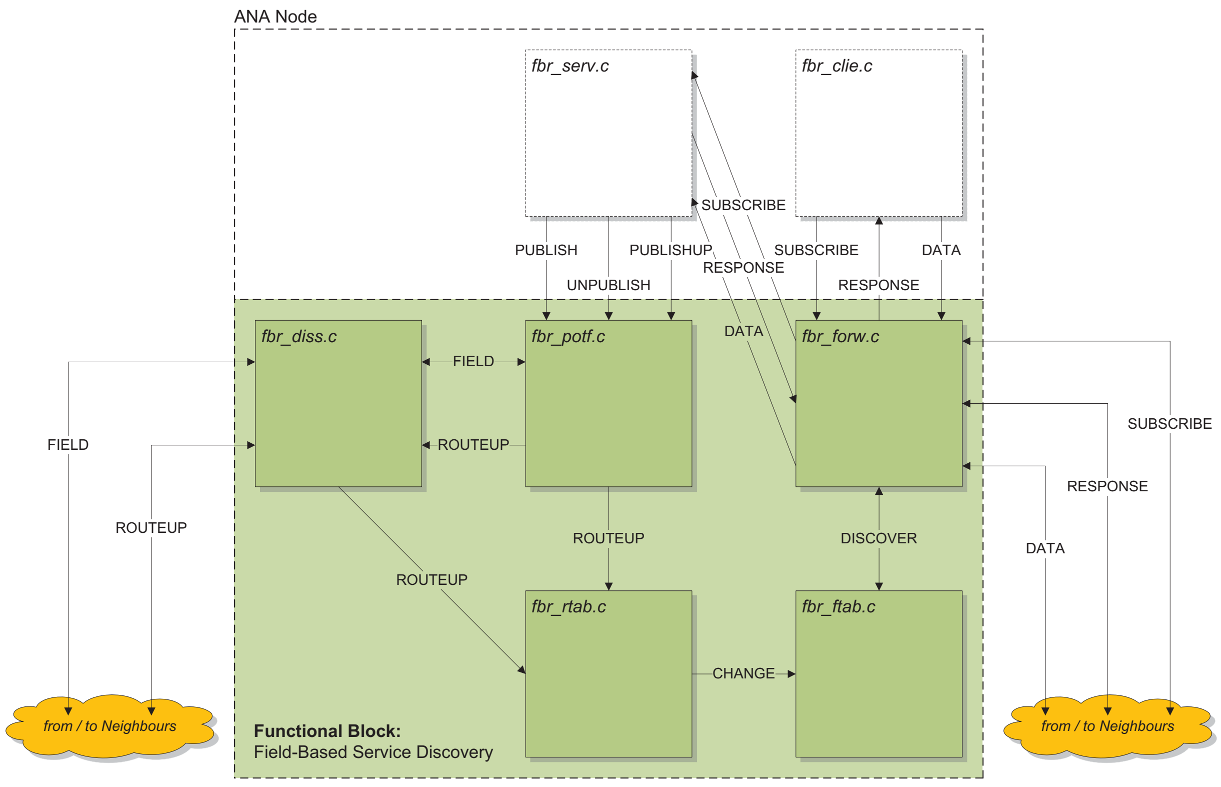


Figure 5.3: Functional block of field-based service discovery in the ANA Playground

5.2.2 Parametrization

The implementation of field-based service discovery offers some ability to tune on different parameters. The parameters and constants all are defined in the header file located at `C/include/fbr.h` in the trunk directory. The header file is included in all bricks involved.

```
#define PUBLISH_TTL 15
#define SUBSCRIBE_TTL 15
#define SERVICE_UPDATE_TIME 90
#define ROUTE_UPDATE_TIME 90
#define GARBAGE_UPDATE_TIME 90
#define GARBAGE_LIFES 2
#define MAX_INTERFACES 10
```

The parameters are mostly self-explanatory. Nevertheless, the description and effect of each parameter can be read in this part. The values offered are default values which have been tested.

- **PUBLISH_TTL:** Defines the time to live for *FIELD* messages. Here, one is able to change the scope of the field resulting from publishing a service instance. If the TTL is over, the *FIELD* message will be dropped.
- **SUBSCRIBE_TTL:** Defines similar to the parameter above the maximum scope of *SUBSCRIBE* messages. If a *SUBSCRIBE* message has to hop over more nodes than specified here, it will be dropped.

- **SERVICE_UPDATE_TIME:** Defines the time in seconds between periodic *PUBLISHUP* messages from the service instances. For now, each service instance has an identical publish update time. If some service instance is rarely supposed to change or even needs to update more often (for example a gateway with the bandwidth declared as CoS), an individual publish update time can be hard-coded.
- **ROUTE_UPDATE_TIME:** Defines the time in seconds between periodic *ROUTEUP* messages.
- **GARBAGE_UPDATE_TIME:** Defines the time in seconds between periodic garbage collecting processes. The garbage collecting process is decreasing the ages of several service type, service instance, potential or route entries in data structures of the bricks starting at the value of GARBAGE_LIFES. The described *PUBLISHUP* and *ROUTEUP* messages respectively reset the age of the belonging entries to the value of GARBAGE_LIFES. If an age is becoming zero, the entry will be deleted.
- **GARBAGE_LIFES:** As described above, this parameter defines the starting value for the ages of service type, service instance, potential and route entries in the data structures of the bricks. Together with the parameter GARBAGE_UPDATE_TIME, one is able to balance the garbage collecting process.
- **MAX_INTERFACES:** This parameter stands for the maximum amount of network interfaces, e.g. Ethernet bricks, for one node. This value is necessary because at a certain point in the program source code, every interface has to be addressed separately.

For the other specifications and data structures which one can find in the header file, refer to the appendix C containing the doxygen documentation of the program source code.

5.2.3 Communication Interfaces

This part describes the important interfaces between the different bricks. Without a clear definition of the interfaces, the architecture can not provide the aspired flexibility. Therefore, one has to picture out the message format of the exchanged dynamic containers. Messages inside a node as well as messages between nodes are XRP encoded. The following explanations want to picture out the XRP message encoding in detail (XRP commands and XRP classes as introduced in section 3.4). In addition, the messages between nodes are shown bit by bit. Obviously, the information provided in this part is fundamental in order to define the interfaces inside a node and between nodes.

5.2.3.1 Inside a Node

- Figure 5.4 shows the XRP messages *PUBLISH*, *UNPUBLISH* and *PUBLISHUP*:

  This is the interface between a service instance and the field-based service discovery compartment.

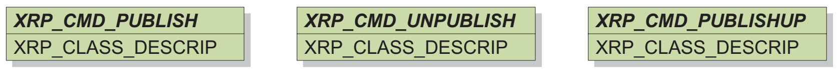


Figure 5.4: XRP messages PUBLISH, UNPUBLISH and PUBLISHUP

- The *PUBLISH* message is encoded as follows:
  - **XRP_CMD_PUBLISH:** Identifies a *PUBLISH* message.
  - **XRP_CLASS_DESCRIP:** Contains the publish description of the service type as well as the point charge or CoS. The format of the XRP_CLASS_DESCRIP class has to look like: `service_type+capacity`, whereas `service_type` stands, as the name implies, for the service type and `capacity` stands for a numerical CoS.
- The *UNPUBLISH* message is encoded as follows:
  - **XRP_CMD_UNPUBLISH:** Identifies an *UNPUBLISH* message.
  - **XRP_CLASS_DESCRIP:** Contains the service type according to the description published previous.
- The *PUBLISHUP* message is encoded as follows:
  - **XRP_CMD_PUBLISHUP:** Identifies a *PUBLISHUP* message.
  - **XRP_CLASS_DESCRIP:** Contains the publish description of the service type as well as the point charge or CoS. The format of the XRP_CLASS_DESCRIP class has to look like: `service_type+capacity`, whereas `service_type` stands, as the name implies, for the service type and `capacity` stands for a numerical CoS.

Note that the *PUBLISHUP* message looks identical to the *PUBLISH* message. Naturally, the CoS can be updated. *UNPUBLISH* messages only need to carry the service type. The field computation brick will delete the service type in order to be able to learn a novel path to another service instance regarding this specific service type. For now, it is only possible to publish one service instance of a specific service type per node (multiple service types are possible) as it can be seen in the implementation section 5.2, the service instance itself does not know its unique service ID.

- In figure 5.5, one can see the XRP messages *FIELD* and *ROUTEUP*:

  These are the field and routing containers which have to be exchanged between neighbour nodes.

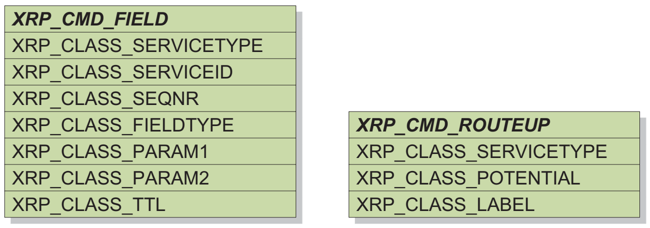


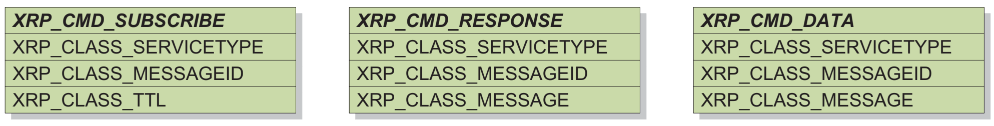
Figure 5.5: XRP messages FIELD and ROUTEUP

- The *FIELD* message is encoded as follows:
  - **XRP_CMD_FIELD:** Identifies a *FIELD* message.
  - **XRP_CLASS_SERVICETYPE:** Describes the type of service.
  - **XRP_CLASS_SERVICEID:** Unique ID for every service instance of a specific service type in order to distinguish between different service instances.
  - **XRP_CLASS_SEQUENCENR:** Contains the sequence number, needed to differentiate old from new messages.
  - **XRP_CLASS_FIELDTYPE:** Contains the field type, e.g. ”electrostatic”.
  - **XRP_CLASS_PARAM1:** A variable of the potential function characterizing the field, e.g. the CoS.
  - **XRP_CLASS_PARAM2:** Another variable of the potential function, e.g. the hop count as a kind of distance metrics.
  - **XRP_CLASS_TTL:** Contains, as the name implies, the TTL for the message. This value is able to restrict the dimensions of the field and thus the publish scope.
- The *ROUTEUP* message is encoded as follows:
  - **XRP_CMD_ROUTEUP:** Identifies a *ROUTEUP* message.
  - **XRP_CLASS_SERVICETYPE:** Contains the service type.
  - **XRP_CLASS_POTENTIAL:** Contains the potential of the sender node according to the specific service type.
  - **XRP_CLASS_LABEL:** Contains the IDP opening an information channel to the sender of the *ROUTEUP* message.

  Note that the IDP entry will be added at reception by the dissemination brick. Thus, a *ROUTEUP* message is sent without the IDP entry. The reason of this behaviour is because in the ANA world each node has to resolve an information channel to another node by itself.

• The XRP messages *SUBSCRIBE*, *RESPONSE* and *DATA* are visible in figure 5.6:

This is the interface between client applications and the field-based service discovery compartment. The messages are payload messages which has to be exchanged between neighbour nodes and are supposed to reach service instances as well as client applications.



XRP_CMD_SUBSCRIBE
XRP_CLASS_SERVICETYPE
XRP_CLASS_MESSAGEID
XRP_CLASS_TTL

XRP_CMD_RESPONSE
XRP_CLASS_SERVICETYPE
XRP_CLASS_MESSAGEID
XRP_CLASS_MESSAGE

XRP_CMD_DATA
XRP_CLASS_SERVICETYPE
XRP_CLASS_MESSAGEID
XRP_CLASS_MESSAGE

Figure 5.6: XRP messages SUBSCRIBE, RESPONSE and DATA

- The *SUBSCRIBE* message is encoded as follows:
  - **XRP_CMD_SUBSCRIBE:** Identifies a *SUBSCRIBE* message.
  - **XRP_CLASS_SERVICETYPE:** Describes the type of service.
  - **XRP_CLASS_REQUESTER:** Contains the requester name, e.g. the description of the client application.

  While leaving the node, a unique message ID for identification needs and a TTL value will be attached to the *SUBSCRIBE* message.

- The *RESPONSE* message is encoded as follows:
  - **XRP_CMD_RESPONSE:** Identifies a *RESPONSE* message.
  - **XRP_CLASS_SERVICETYPE:** Contains the service type.
  - **XRP_CLASS_MESSAGEID:** Represents the message ID of the initial *SUBSCRIBE* message in order to find the feedback channel.
  - **XRP_CLASS_MESSAGE:** Contains some kind of response for the client application.
- The *DATA* message is encoded as follows:
  - **XRP_CMD_DATA:** Identifies a *DATA* message.
  - **XRP_CLASS_SERVICETYPE:** Contains the service type.
  - **XRP_CLASS_MESSAGEID:** Represents the message ID of the initial *SUBSCRIBE* message in order to identify the discovered path to the service instance again.
  - **XRP_CLASS_MESSAGE:** Holds the payload sent to the service instance.

Note that the *RESPONSE* message and the *DATA* message are encoded identical. Nevertheless, the differentiation is necessary to be able to separate at a certain node if a message is traveling towards a service instance (*DATA* message) or towards a client (*RESPONSE* message). One can read more on this in the design section 5.1.

- Figure 5.7 pictures out the XRP messages *CHANGE* and *DISCOVER*:

  This is the interface to the forwarding table which is used by the routing table brick as well as the forwarding brick.

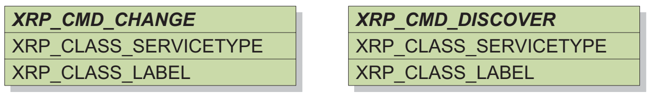


Figure 5.7: XRP messages CHANGE and DISCOVER

- The *CHANGE* message is encoded as follows:
  - **XRP_CMD_CHANGE:** Identifies a *CHANGE* message.
  - **XRP_CLASS_SERVICETYPE:** Describes the type of service.
  - **XRP_CLASS_LABEL:** Contains the IDP for the next hop of the specific service type. If the next service instance is located locally, the XRP_CLASS_LABEL should contain "KVR" in order to tell the forwarding brick to stop forwarding and look for the service instance in the KVR of the MINMEX.
- The *DISCOVER* message is encoded as follows:
  - **XRP_CMD_DISCOVER:** Identifies a *DISCOVER* message.
  - **XRP_CLASS_SERVICETYPE:** Contains the service type.

  Because the forwarding table holds only one entry for each service type, it is the only information the forwarding table needs to know in order to discover the next hop.

5.2.3.2 Between Nodes

In this part, the XRP messages exchanged between nodes are shown bit by bit. The XRP messages all start with the XRP command, three bytes in length. Then, the number of attached arguments is given with four bytes. Finally, each argument is mentioned first with the XRP class, again three bytes in length, then with the size of the actual value, 4 bytes in length and in the end, the value itself. Refer to 3.4 for a more detailed discussion about XRP message formats.

- Figure 5.8 shows the XRP messages (containers) *FIELD* and *ROUTEUP* exchanged between nodes:

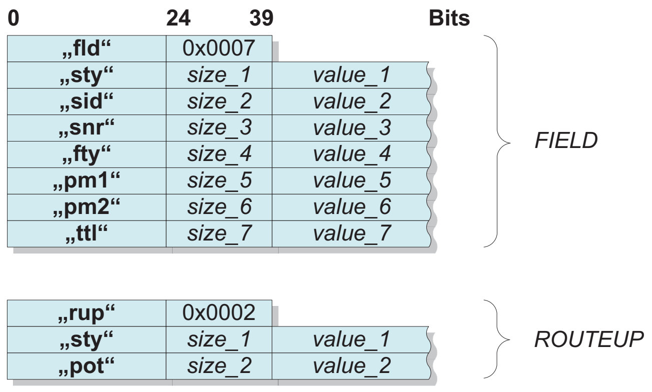


Figure 5.8: XRP messages FIELD and ROUTEUP exchanged between nodes

  – The *FIELD* message is encoded in the same way like inside a node:
    - **”fld”:** Represents XRP_CMD_FIELD.
    - **”sty”:** Represents XRP_CLASS_SERVICETYPE.
    - **”sid”:** Represents XRP_CLASS_SERVICEID.
    - **”snr”:** Represents XRP_CLASS_SEQUENCENR.
    - **”fty”:** Represents XRP_CLASS_FIELDTYPE.
    - **”pm1”:** Represents XRP_CLASS_PARAM1.
    - **”pm2”:** Represents XRP_CLASS_PARAM2.
    - **”ttl”:** Represents XRP_CLASS_TTL.
  – The *ROUTEUP* message is encoded in the same way as inside a node, except the missing IDP class which will be attached at reception:
    - **”rup”:** Represents XRP_CMD_ROUTEUP.
    - **”sty”:** Represents XRP_CLASS_SERVICETYPE.
    - **”pot”:** Represents XRP_CLASS_POTENTIAL.

- The XRP messages *SUBSCRIBE*, *RESPONSE* and *DATA* exchanged between nodes, one can see in figure 5.9:

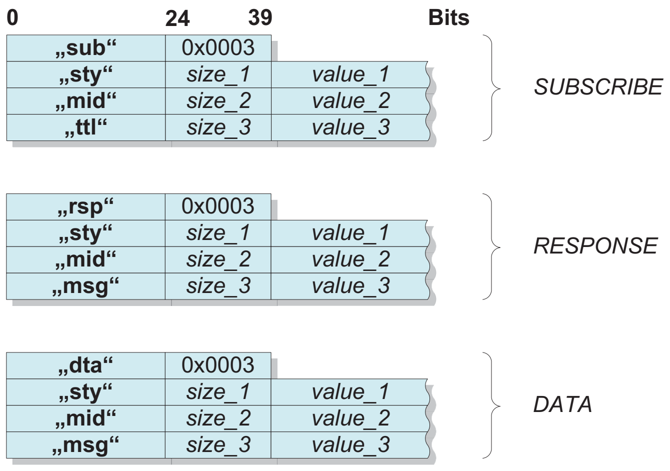


Figure 5.9: XRP messages SUBSCRIBE, RESPONSE and DATA exchanged between nodes

- The *SUBSCRIBE* message is encoded as follows:
  - **"sub":** Represents XRP_CMD_SUBSCRIBE.
  - **"sty":** Represents XRP_CLASS_SERVICETYPE.
  - **"mid":** Represents XRP_CLASS_MESSAGEID.
  - **"ttl":** Represents XRP_CLASS_TTL.
- The *RESPONSE* message is encoded identical like inside a node:
  - **"rsp":** Represents XRP_CMD_RESPONSE.
  - **"sty":** Represents XRP_CLASS_SERVICETYPE.
  - **"mid":** Represents XRP_CLASS_MESSAGEID.
  - **"msg":** Represents XRP_CLASS_MESSAGE.
- The *DATA* message is encoded in the same way as inside a node:
  - **"dta":** Represents XRP_CMD_DATA.
  - **"sty":** Represents XRP_CLASS_SERVICETYPE.
  - **"mid":** Represents XRP_CLASS_MESSAGEID.
  - **"msg":** Represents XRP_CLASS_MESSAGE.

5.2.4 The Bricks

This part deals with each brick separately. It gives a description of the purpose of every brick and additionally a well-explained illustration of the internal data processing. If one wants to know more about certain data processing, refer to the comments in the program source code and to the doxygen documentation in the appendix C.

5.2.4.1 Field Computing Brick

The field computing brick is the heart of the field-based service discovery architecture and as the name implies, it handles with potential functions and computes the potentials of the nodes. If a service instance wants to publish itself, it has to communicate with the field computing brick. *FIELD* messages from neighbour nodes are also directed to the field computing brick. Thus, the field computing brick holds data structures for local service instances and for all service type potentials traceable in the compartment. Furthermore, it sends periodic *PUBLISHUP* and *ROUTEUP* messages.

To become part of the architecture, there are four different kind of data the field computing brick can receive during lifetime:

- PUBLISH messages
- UNPUBLISH messages
- PUBLISHUP messages
- FIELD messages

Each message is received at the same data receive IDP published in the KVR of the MINMEX.

Receiving a *PUBLISH* message

The instructions executed by receiving a *PUBLISH* message from a local service instance are depicted in figure 5.10:

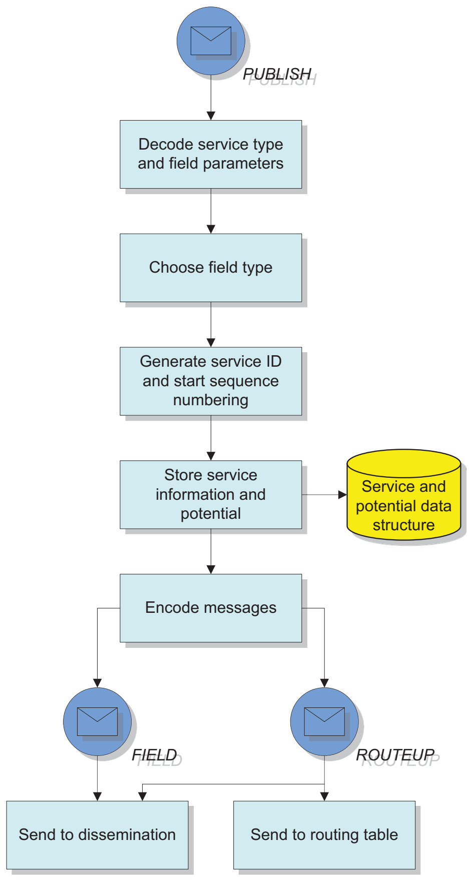

PUBLISH
Decode service type and field parameters
Choose field type
Generate service ID and start sequence numbering
Store service information and potential
Service and potential data structure
Encode messages
FIELD
ROUTEUP
Send to dissemination
Send to routing table

Figure 5.10: Data processing of the field computing brick receiving a PUBLISH message

1. The service type and some kind of point charge, e.g. the CoS are decoded.
2. An appropriate field type is chosen for this specific service type. For now, it is per default always an ”electrostatic” field.
3. Then, a unique service ID will be generated for this specific service instance. This is done by means of the random number generator. The unique ID results as a six digit random number, whereas the seed of the random number generator is depending on the system time.
4. Additionally, the sequence counting is starting. For now, it is per default starting at ”100”.
5. Now, the brick is able to store the information in the specific data structures. The brick holds three types of data structures (appendix C):

   (a) All the information about the local service instances, necessary to encode *PUBLISHUP* messages in the further lifetime of the node.
   (b) Every service ID and sequence number of all service instances situated in the compartment, necessary to identify service instances when receiving *FIELD* messages from neighbour nodes.
   (c) The computed potentials depending on the field type for each service type situated in the compartment, necessary to store the potentials for the local node and to encode *ROUTEUP* messages in the further lifetime of the node. The potential for a local service instance is per definition infinite.

6. Afterwards, the brick encodes two types of XRP messages.
   (a) On the one hand, the brick encodes a *FIELD* message and forwards it to all neighbour nodes in order to distribute the field information. This message is sent only to the dissemination brick whereas on the other hand,
   (b) a *ROUTEUP* message is encoded with the computed potential and sent to the dissemination brick and to the routing table brick. The routing table needs to know the local potentials in order to be able to stop forwarding *SUBSCRIBE* and *DATA* messages from a client application.

Receiving an *UNPUBLISH* message

By receiving an *UNPUBLISH* message from a local service instance, the steps taken according to figure 5.11 are:

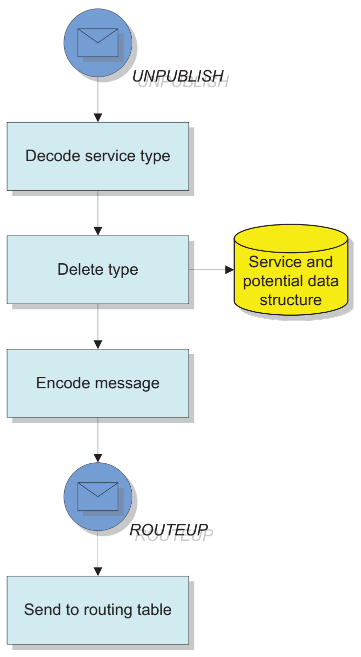


Figure 5.11: Data processing of the field computing brick receiving an UNPUBLISH message

1. Foremost, the service type is decoded.
2. Afterwards, the brick is able to delete the information in the specific data structures (appendix C):
   (a) The information about the local service instance will be cleaned up as well as
   (b) the potential for this specific service type. The potential for the service type needs to be unloaded in order to learn a novel potential dissimilar from infinite.
3. Afterwards, the brick encodes a *ROUTEUP* message for the routing table brick. The routing table needs to know about this erasure event so that it is able to delete the specific route entry and to be in a position telling the forwarding table a novel route towards another service instance.

Receiving a *PUBLISHUP* message

When the brick receives a *PUBLISHUP* message from a local service instance, as can be seen in figure 5.12, the following instructions are performed:

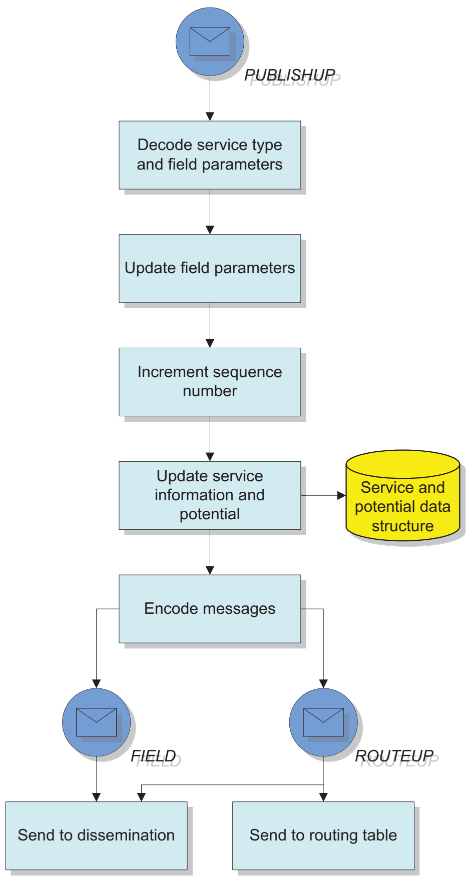

PUBLISHUP
Decode service type and field parameters
Update field parameters
Increment sequence number
Update service information and potential
Service and potential data structure
Encode messages
FIELD
ROUTEUP
Send to dissemination
Send to routing table

Figure 5.12: Data processing of the field computing brick receiving a PUBLISHUP message

1. The service type and some kind of point charge, e.g. the CoS are decoded. The new CoS may be different from the old one.
2. The sequence number is increased by one.
3. Now, the brick is able to update the information in the specific data structures (appendix C).
4. Afterwards, the brick encodes and sends two types of XRP messages analogously to step six above when receiving a *PUBLISH* message.

Receiving a *FIELD* message

As soon as a *FIELD* message from a remote node arrives like in figure 5.13, the proximate details are run:

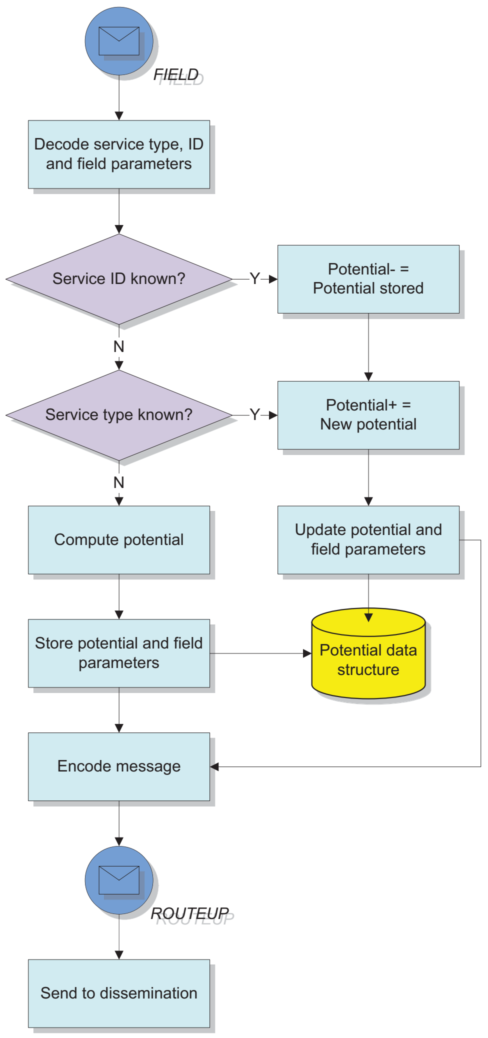

FIELD
Decode service type, ID and field parameters
Service ID known?
Y
Potential- = Potential stored
N
Service type known?
Y
Potential+ = New potential
N
Compute potential
Update potential and field parameters
Store potential and field parameters
Potential data structure
Encode message
ROUTEUP
Send to dissemination

Figure 5.13: Data processing of the field computing brick receiving a FIELD message

1. First of all, the service type, service ID and the field parameters are decoded.
2. When the service ID, e.g. the service instance is known by the brick, the stored total potential for this service type will be altered. First, the old potential quotient of this specific service instance is subtract from the total potential, then the new potential quotient is added.
3. When the service instance is alien for the node but the service type is known, the summed up potential will be raised with the new potential quotient from the recent service instance.
4. Otherwise, neither the service type nor the service instance is familiar, a brand-new potential from scratch will be computed depending on the field type and field parameters.
5. Now, the brick is able to store the information in the specific data structures (appendix C), similar as the message would have been the result of a local service instance. In this case, only two data structures have to be modified, because the local service instance data structure has no influence.
6. Afterwards, the brick encodes one type of XRP message. A *ROUTEUP* message is encoded with the novel, total potential and sent to the dissemination brick.

Besides the instructions triggered by an event as described above, the brick owns timer functions in order to run several periodic updates. The information in the local service instance data structure are used to encode periodic *PUBLISHUP* messages whereas the information in the potential data structure are required to encode periodic *ROUTEUP* messages for the neighbour nodes. Last but not least, there is also a garbage collecting function existing which tidies up with data entries, suffering from missing updates.

5.2.4.2 Routing Table Brick

The routing table brick is in charge of holding the routing table. Therefore, it is responsible to know all potentials of the neighbour nodes. Each neighbour node possesses an entry for each service type traceable in the compartment. The local service instances are also entered because the table is updating the forwarding table and needs to know when a forwarding process achieved its aim.

In order to become a part of the functional block, there is just one kind of data the routing table brick will receive during lifetime:

- *ROUTEUP* messages

Each message is received at the same data receive IDP published in the KVR of the MINMEX.

Receiving a *ROUTEUP* message

The instructions executed by receiving a *ROUTEUP* message are shown in figure 5.14:

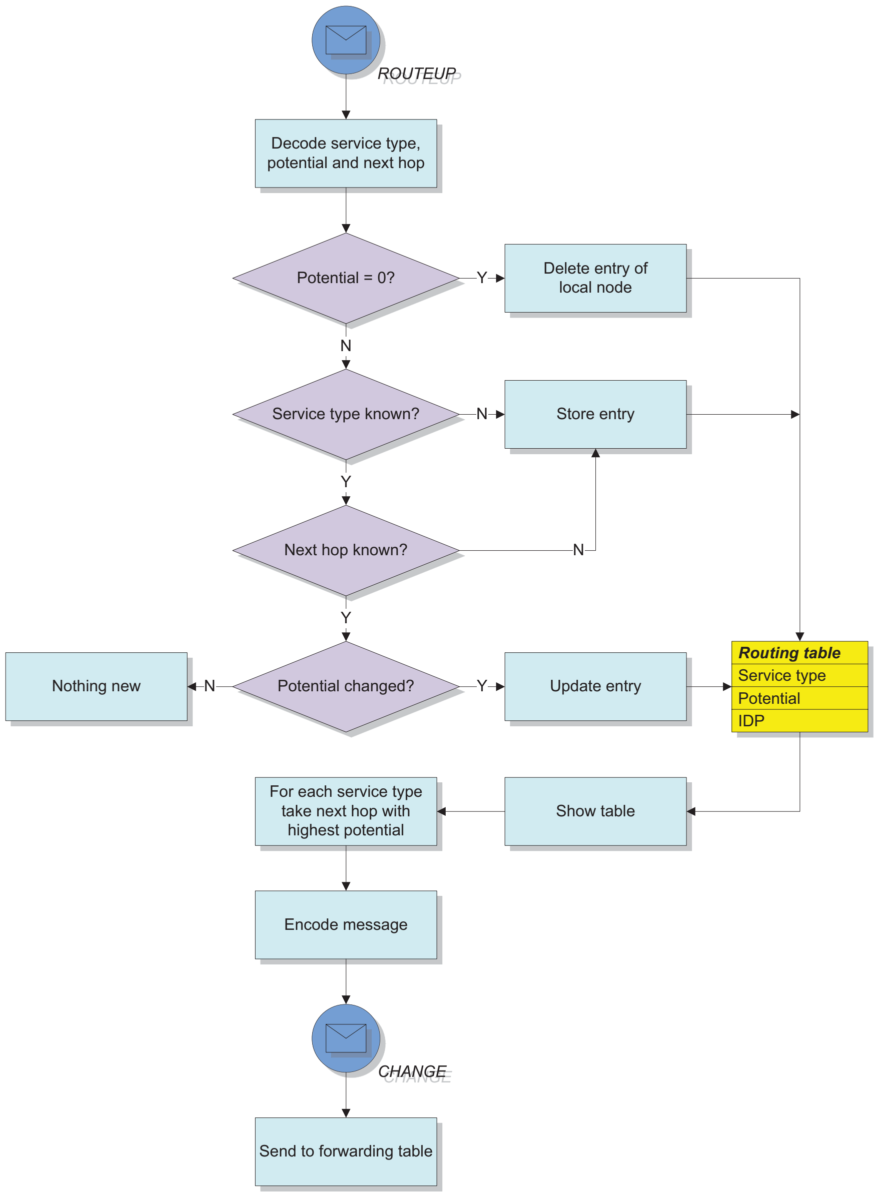


Figure 5.14: Data processing of the routing table brick receiving a ROUTEUP message

1. The service type, the belonging potential and the next hop are decoded.
2. When the potential is zero, the *ROUTEUP* message is coming from the field computing brick, resulting from unpublishing a local service instance. The entry of this specific service instance needs to be deleted.

3. Else, if the service type is unknown or the service type is known but the next hop is foreign, the entry will be stored. In this case, a new neighbour node with a belonging potential becomes well-established.

4. When the next hop is familiar as well, the brick has to compare the stored potential with the new one. If there is a difference, the entry needs to be updated because the potential concerning a service type on a neighbour node has changed. Should there be nothing new, the message can be dropped.

5. Now, as the routing table has changed, it is shown on the brick output. Additionally, the brick updates the forwarding table. For each service type, the entry with the highest potential is taken and sent to the forwarding table because routing is done along the steepest gradient of the field. Therefore, the routing table brick encodes for each service type one *CHANGE* message and sends all these messages to the forwarding table brick.

Besides the instructions triggered by an event as described above, the brick owns a timer function in order to run periodic updates for the forwarding table. Moreover, there is also a garbage collecting function existing which tidies up with route entries suffering from missing updates.

5.2.4.3 Dissemination Brick

The dissemination brick is responsible for distributing the information such as *FIELD* messages and *ROUTEUP* messages to the neighbour nodes and acts as a doorman to pick up the same information from other nodes. In order to reduce overhead, the dissemination brick holds a data structure for service IDs and sequence numbers as well. Therefore, it is able to identify old messages which will not be delivered further.

To become part of the architecture, there are two different kind of data the dissemination brick can receive during lifetime:

- *FIELD* messages
- *ROUTEUP* messages

Both message types are sent by the local field computing brick as well as by neighbour nodes. There exist two different data receive IDPs in order to separate message from neighbour nodes and from a local brick. The data receive IDP for the neighbour nodes is published in the Ethernet brick and the one for the field computing brick is published in the KVR of the MINMEX.

Receiving a *FIELD* message from the field computing brick

The instructions executed by receiving a *FIELD* message from the field computing brick are depicted in figure 5.15:

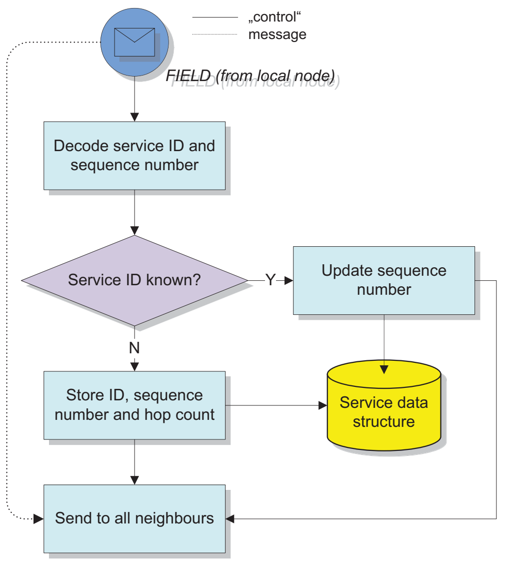


Figure 5.15: Data processing of the dissemination brick receiving a FIELD message from the field computing brick

1. The service ID and the sequence number are decoded. This is done because these information can be stored and used to reject own *FIELD* messages coming back. The information are stored in a service data structure (appendix C). In addition, the hop count is also stored in order to be able to choose the appropriate message, if two messages contain the same service ID and an identical sequence number.
2. Finally, the message is distributed to all neighbour nodes. This is achieved with the underlying data link compartment, e.g. the Ethernet compartment. The dissemination brick opens an information channel to the neighbour nodes by resolving the targets with help of AL2:

   ```
   broadcastIDP = anaL2_resolve(ETHERNET_LABEL, (void *) "*", 2,
   (void *) "fbr_diss", 9, 'b', NULL, 0);
   ```

Receiving a *ROUTEUP* message from the field computing brick

When the brick receives a *ROUTEUP* message from the field computing brick, the following instructions are performed, displayed in figure 5.16:

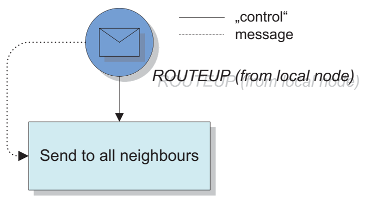


Figure 5.16: Data processing of the dissemination brick receiving a ROUTEUP message from the field computing brick

1. Simply, the *ROUTEUP* message is forwarded to all neighbour nodes. This distribution is done by means of the underlying data link compartment in the same way as described above.

Receiving a *FIELD* message from a neighbour node

By receiving a *FIELD* message from a service instance located on a remote node, figure 5.17 exhibits:

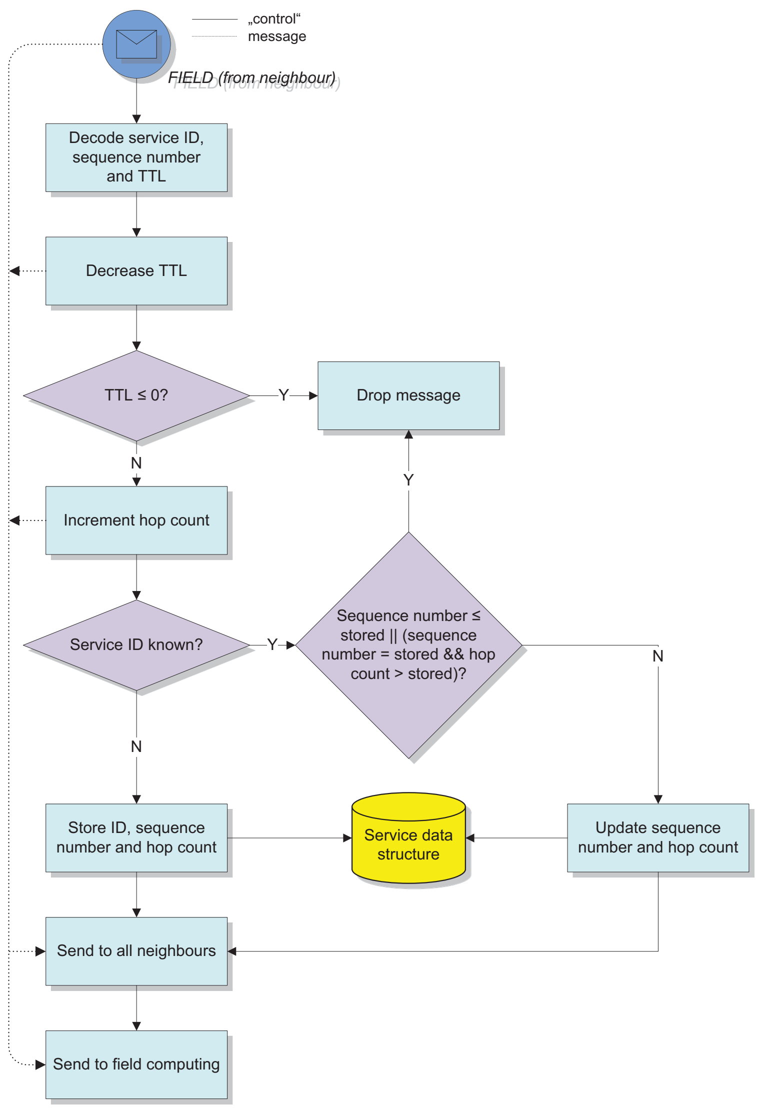

„control“
message
FIELD (from neighbour)
Decode service ID, sequence number and TTL
Decrease TTL
TTL ≤ 0?
Y
Drop message
N
Y
Increment hop count
Service ID known?
Y
Sequence number ≤ stored || (sequence number = stored && hop count > stored)?
N
N
Store ID, sequence number and hop count
Service data structure
Update sequence number and hop count
Send to all neighbours
Send to field computing

Figure 5.17: Data processing of the dissemination brick receiving a FIELD message from a neighbour node

1. The service ID, the sequence number and the TTL are decoded.
2. The TTL is decremented by one. If the TTL is over, the message will be dropped.
3. Afterwards, the hop count is incremented by one.
4. If the service ID is not known, the brick has just learned a new service instance and stores the relevant information in a data structure (appendix C). If the service ID is

known, the brick needs to decide if the message is newsworthy. This is achieved by looking at the sequence number and the hop count. If the sequence number is higher than the one stored or if the sequence number is equal but the hop count is lower than the one stored, the message is relevant as it is either a brand-new message or it traveled along fewer nodes. If nothing from the above mentioned applies, the message is old or uninteresting and will be dropped as well.

5. When the brick is dealing with an important message, the message, including the novel TTL and the novel hop count, is forwarded to the field computing brick and to all neighbour nodes by means of the underlying data link compartment in the same way as described above.

Receiving a *ROUTEUP* message from a neighbour node

As soon as a *ROUTEUP* message from a neighbour node arrives, the proximate details from figure 5.18 are run:

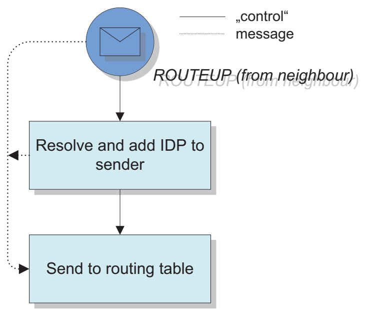


Figure 5.18: Data processing of the dissemination brick receiving a ROUTEUP message from a neighbour node

1. The *ROUTEUP* message coming from a neighbour node is incomplete. The IDP in order to communicate with the neighbour node is missing because each node needs to resolve its IDPs themselves. Thus, if the node wants a valid IDP to the sender, it has to resolve the sender with the underlying data link compartment.
2. Afterwards, the IDP is added to the encoded message and the complete *ROUTEUP* message will be forwarded to the routing table brick.

Furthermore, a garbage collecting function is also existing which tidies up with data entries suffering from missing updates.

5.2.4.4 Forwarding Brick

The forwarding brick is responsible for forwarding payload. Before the payload is sent actually, client applications are able to discover a service instance with *SUBSCRIBE* messages. Moreover, the forwarding brick handles the way how *RESPONSE* messages are sent back to the client application, after a service instance has been found successfully or a *DATA* message has been received. The forwarding brick is the counterpart of the IP encapsulation brick with the slight difference that there is no analogous encapsulation existing.

In order to become part of the functional block there are three different kind of data the forwarding brick can receive during lifetime:

- *SUBSCRIBE* messages
- *RESPONSE* messages
- *DATA* messages

All message types are either sent by a local client application, service instance or neighbour nodes. There exist two different data receive IDPs in order to separate message from neighbour nodes and from a local brick. The data receive IDP for the neighbour nodes is published in the Ethernet brick and the one for local bricks is published in the KVR of the MINMEX.

Receiving a *SUBSCRIBE* message from a local client application

The instructions executed by receiving a *SUBSCRIBE* message from a local client application presents figure 5.19:

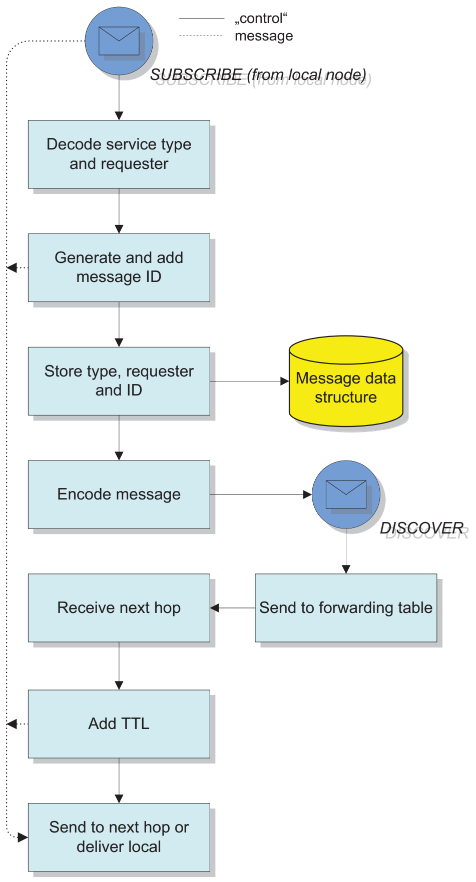

„control"
message
SUBSCRIBE (from local node)
Decode service type and requester
Generate and add message ID
Store type, requester and ID
Message data structure
Encode message
DISCOVER
Receive next hop
Send to forwarding table
Add TTL
Send to next hop or deliver local

Figure 5.19: Data processing of the forwarding brick receiving a SUBSCRIBE message from a local client application

1. The desired service type and the requester name are decoded. The requester name is the description of the client application. It is necessary in order to successfully deliver a further *RESPONSE* message.

2. A unique message ID will be generated for this specific *SUBSCRIBE* message. This is done by means of the random number generator. The unique ID results as a eight digit random number whereas the seed of the random number generator is depending on the system time. The message ID needs to be added to the *SUBSCRIBE* message.

3. Now, the brick stores the service type, requester name and message ID in a message data structure (appendix C). This storage is necessary to deal with further messages concerning the same message ID, for example a *RESPONSE* message from a service instance.

4. One type of XRP message is encoded. The brick sends a *DISCOVER* message to the forwarding table brick because it needs to know the next hop regarding the specific service type.
5. As soon as the answer from the forwarding table is received, the brick knows where to send the message. Depending on the answer, the next hop is either a neighbour node or a local brick if the service type possesses an instance on the same node. Before sending, the TTL is added.

Receiving a *SUBSCRIBE* message from a neighbour node

As soon as a *SUBSCRIBE* message from a neighbour node arrives, the proximate details indicated in figure 5.20 are run:

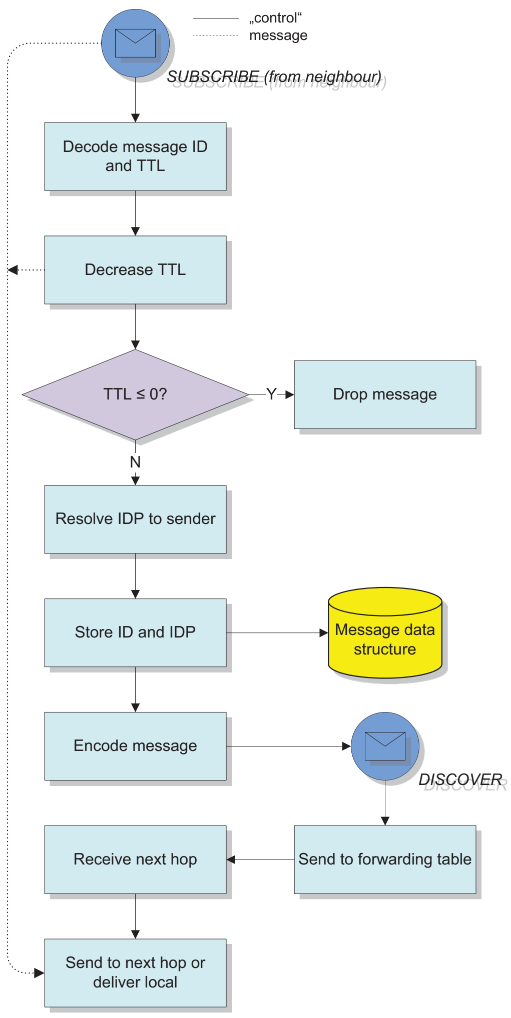


Figure 5.20: Data processing of the forwarding brick receiving a SUBSCRIBE message from a neighbour node

1. First of all, the message ID and the TTL are decoded.
2. The TTL is decremented. If the TTL is over, the message will be dropped.
3. Because each node needs to resolve its IDPs themselves, an IDP to the sender is resolved. This step is necessary in order to store a feedback channel for further *RESPONSE* messages. This is only one way to open a feedback channel. The *RESPONSE* messages take exactly the same path back to the client applications. Other possibilities are imaginable as described in the design section 5.1. One is free to exchange the forwarding brick to handle feedback channels regarding the own requirements.
4. Now, the brick stores the information in the message data structure (appendix C). This storage is necessary in order to deal with further messages concerning the same message ID, for example a *RESPONSE* message from a service instance.
5. Next, one type of XRP message is encoded. The brick sends a *DISCOVER* message to the forwarding table brick because it needs to know the next hop regarding the specific service type.
6. As soon as the answer from the forwarding table is received, the brick knows where to send the message. Depending on the answer, the next hop is either a neighbour node or a local brick if the service type possesses an instance on the same node.

Receiving a *RESPONSE* message from a local service instance

By receiving a *RESPONSE* message from a local service instance, the steps taken shows figure 5.21:

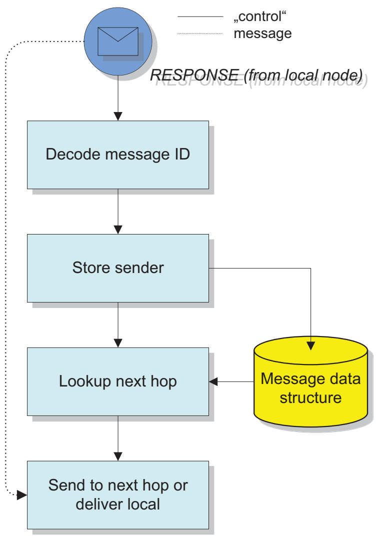


Figure 5.21: Data processing of the forwarding brick receiving a RESPONSE message from a local service instance

1. The message ID is decoded.
2. Now, the sender of the message is stored in the message data structure (appendix C) in order to be able to deliver further *DATA* messages.

3. The brick looks up the next hop in the message data structure (appendix C) because the next hop has been stored as the according *SUBSCRIBE* message passed the node.

4. Depending on the result, the next hop is either a neighbour node or a local brick if the client application sits on the same node.

Receiving a *RESPONSE* message from a neighbour node

When the brick receives a *RESPONSE* message from a neighbour node, the following instructions are performed, illustrated in figure 5.22:

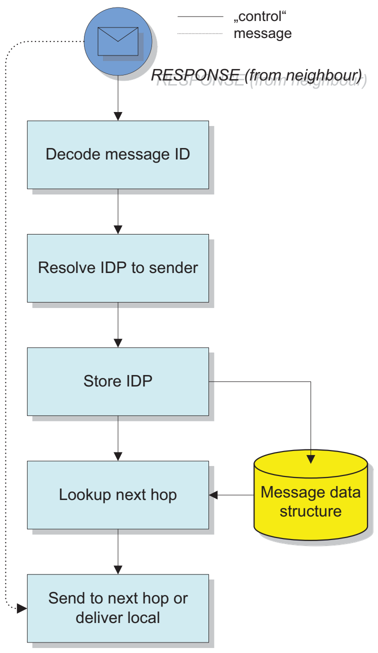


Figure 5.22: Data processing of the forwarding brick receiving a RESPONSE message from a neighbour node

1. The message ID is decoded.

2. An IDP to the sender is resolved to be able to deliver further *DATA* messages concerning this specific message ID.

3. Now, the brick stores the information in the message data structure (appendix C). This storage is necessary in order to deal with further messages concerning the same message ID as described at step two.

4. One type of XRP message is encoded. The brick sends a *DISCOVER* message to the forwarding table brick because it needs to know the next hop regarding the specific service type.

5. As soon as the answer from the forwarding table is received, the brick knows where to send the message. Depending on the answer, the next hop is either a neighbour node or a local brick, if the client application sits on the same node.

Receiving a *DATA* message

Figure 5.23 presents the instructions executed by receiving a *DATA* message (either from a local client or from a neighbour node):

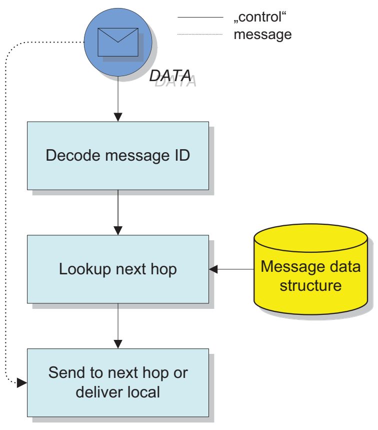


Figure 5.23: Data processing of the forwarding brick receiving a DATA message

1. The message ID is decoded.
2. Then, the brick looks up the next hop in the message data structure (appendix C) because the next hop has been stored as the according *SUBSCRIBE* message with the identical message ID passed the node.
3. Depending on the result, the next hop is either a neighbour node or a local brick if the service instance sits on the same node.

Note that after a successful discovery of a service instance, the message ID is known in all nodes building the discovered path. Therefore, the message ID becomes a communication ID representing an information channel between a specific client application and a specific service instance (including a feedback channel regarding the implementation of this Master Thesis).

5.2.4.5 Forwarding Table Brick

The forwarding table brick holds the forwarding table. Therefore, it is responsible to know the next hops in the forwarding process regarding a specific service type. Each service type traceable in the compartment possesses an entry in the forwarding table, including an IDP to the current next hop.

To become part of the architecture, there are two kind of data the forwarding table brick will receive during lifetime:

- *CHANGE* messages
- *DISCOVER* messages

Each message is received at the same data receive IDP, published in the KVR of the MIN-MEX.

Receiving a *CHANGE* message

The instructions executed by receiving a *CHANGE* message from the routing table brick are displayed in figure 5.24:

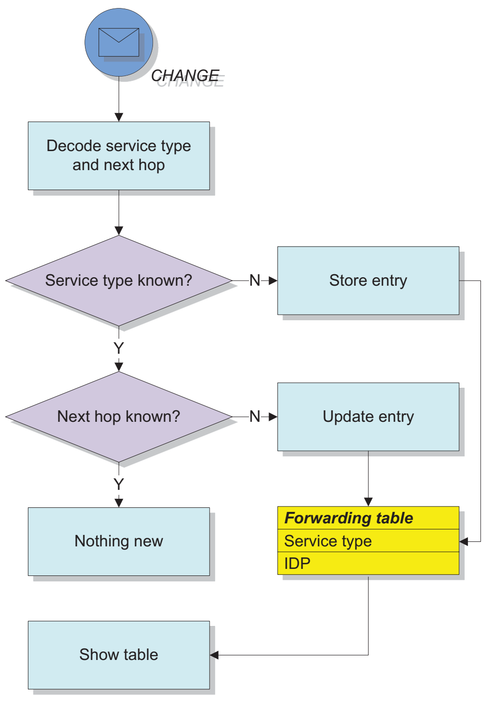


Figure 5.24: Data processing of the forwarding table brick receiving a CHANGE message

1. First, the service type and the next hop are decoded.
2. If the service type is unknown, the entry is stored. In this case, a novel service type with the belonging next hop becomes well-established.
3. When the service type is known but the next hop differs from the one stored, the entry for this specific service type is updated. In this case, the field distribution for this specific service type changed and the node learns a novel route to an optimal service instance. Otherwise, nothing is new and the message will be dropped.
4. If the forwarding table changed, it is shown on the brick output.

Receiving a *DISCOVER* message

Figure 5.25 shows the instructions performed as soon as the brick receives a *DISCOVER* message from the forwarding brick:

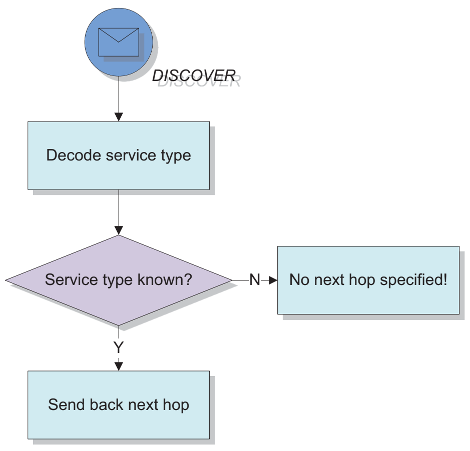


Figure 5.25: Data processing of the forwarding table brick receiving a DISCOVER message

1. The service type is decoded.
2. If the service type is known, the forwarding table brick sends back the belonging next hop. Otherwise, no next hop for this specific service type is known (the node has not learned a route yet) and the message will be dropped.

Moreover, a garbage collecting function is existing which tidies up with route entries, suffering from missing updates.

5.2.4.6 Sample Service Brick

This Master Thesis used a sample service brick and a sample client brick in order to implement and validate the architecture. The sample service brick represents a specific service instance of a certain service type. To sum it up, the sample service brick does the following:

- Publishes the name of the desired service type in the KVR of the MINMEX.
- Sends a *PUBLISH* message to the field computing brick.
- Displays the received *SUBSCRIBE* messages.
- Sends back a *RESPONSE* message containing some text upon receiving a *SUBSCRIBE* message.
- Displays the received *DATA* messages.
- Sends periodic *PUBLISHUP* messages to the field computing brick.
- Unpublishes itself upon brick exit.

One is able to choose the service type and the point charge, e.g. the CoS of the service instance at runtime by typing auxiliary command line arguments:

```
$ ./bin/fbr_serv -n MINMEX_CONTROL_GATE -a SERVICE_TYPE -a CoS
```

5.2.4.7 Sample Client Brick

The sample client brick represents a specific client application which intends to discover a specific service type. The service instance found will be the sample service brick mentioned above. To sum it up, the sample service brick does the following:

- Publishes its description as requester name in the KVR of the MINMEX.
- Sends a *SUBSCRIBE* message containing the desired service type to the forwarding brick.
- Displays the received *RESPONSE* message.
- Sends back a *DATA* message containing some text upon receiving a response message.

One is able to choose the service type to discover at runtime by typing a auxiliary command line argument:

```
$ ./bin/fbr_clie -n MINMEX_CONTROL_GATE -a SERVICE_TYPE
```

5.3 Validation

This section demonstrates the validation of the implementation of field-based service discovery for ANA. At the beginning, it explains the environment which has been used. Afterwards, four different, crucial scenarios have been tested. Field assembly (1) shows the basic field assembly and distribution process of the compartment, after publishing one service instance. The scenario of multiple service instances (2) exhibits the same situation but with two service instances of the same service type published. More than one service type (3), resulting in different field overlays is indicated in the multiple fields part. Finally, the last part of the validation section demonstrates what happens if some node loses the ability to communicate (4).

5.3.1 Setup

For validating an implementation like field-based service discovery for ANA, a testbed network is mandatory. This Master Thesis used the TIK testbed[1] in order to configure an arbitrary topology of ten nodes running ANA.

[1]A collection of hosts attached to both wired Ethernet and wireless LAN interfaces [39]

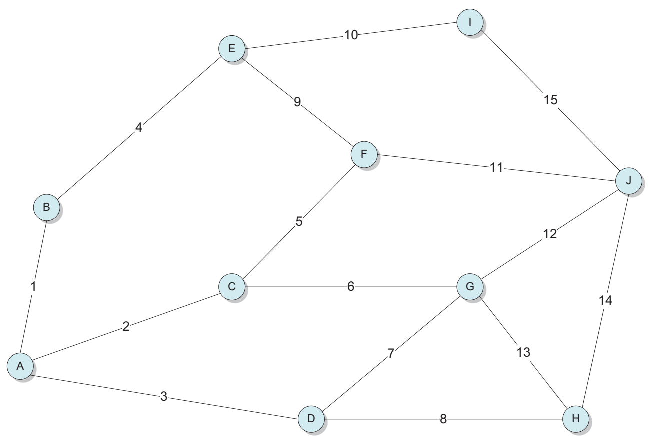


Figure 5.26: Network topology for validating the field-based service discovery implementation for ANA

The nodes all are located in the same Ethernet segment. The topology is realized with the support of virtual links. Virtual links for ANA are easily configurable with the virtual link brick [4], situated in the ANA core software. The instructions how to run the field-based service discovery compartment in ANA are available in the appendix B. The trick is to appoint each connection between two desired nodes with one virtual link ID. The IDs used for this validation topology (1 – 15) and the nodes carrying the names A – J are shown in figure 5.26.

To validate miscellaneous scenarios, one can start sample service bricks and sample client bricks on certain nodes and observe the outputs of all nodes building the compartment. The important brick outputs are on the one hand, the routing and forwarding table entries as well as the computed field potential for one node, regarding the field assembly and the routing process. On the other hand, the outputs of the sample service and sample client bricks as well as the output of the forwarding brick are interesting in order to validate message reception and the path which has been taken.

Remember: For now, every field type chosen by the field computing brick is ”electrostatic”. Therefore, the potential function to compute the potential in a node looks as follows (section 2.5):

$$Potential = \frac{CoS}{Amount\,of\,hops\,to\,service\,instance}$$

All effects observed in the network by taking some action depending on the scenarios are displayed in illustrations (figure 5.27 – 5.30). The illustration shows the validating topology with all nodes again. Forthright above the nodes, one can check the potential of the node for specific service types. Underneath the nodes, the routing table is displayed containing an entry for each neighbour node and service type in the following format:

```
service type | potential | next node
```

The entry with the highest potential for each service type, hence the one that is sitting in the forwarding table, is highlighted in bold. Additionally, the path of a *SUBSCRIBE* message from a client application towards a service instance is highlighted in red.

5.3.2 Field Assembly

In this scenario one service instance is started. The interesting points are the field assembly and the path from a client application to the service instance which the compartment discovered. As one can see in figure 5.27, the service instance is started on node G with service type ”gateway” and a CoS of ”12”. The resulting field potentials for all nodes are shown in table 5.1:

Node	*Potential for service type ”gateway”*
A	6
B	4
C	12
D	12
E	4
F	6
G	infinite
H	12
I	6
J	12

Table 5.1: Computed field potentials after publishing one service instance

After publishing a ”gateway” on node G, a client application is started on node A which tries to discover the ”gateway”. There is no need to wait a long time for the complete field assembly and routing information exchange because these processes are triggered by the publishing event. By reason of the route entries as a result of the routing process one can observe the *SUBSCRIBE* message from the client application traveling first to node D and then to its final destination, node G. On node G one is able to see on the output of the sample service brick:

```
--> subscribe message received

fbr_serv: message-id: 91706962
fbr_serv: successful register_callback
fbr_serv: anaL2_resolve: Resolve Success
fbr_serv: response message sent to forwarding brick
```

Apparently, the client discovered the service instance successfully. On the output of the sample client brick on node A one can observe:

```
--> response message received

fbr_clie: message-id: 91706962
fbr_clie: message: Hi! I received your subscribe message!
fbr_clie: data message sent
```

Obviously, the client application received a *RESPONSE* message back from the service instance and sends out again some data. Thus, on the sample service brick on node G one can see:

```
--> data message received

fbr_serv: message-id: 91706962
fbr_serv: message: Hi! Now, I know you!
```

In conclusion, the field assembly process upon publishing a service instance on node G was successful on all nodes. Furthermore, a client application effectually discovered the service instance, hence node A and node G exchanged the message types as expected.

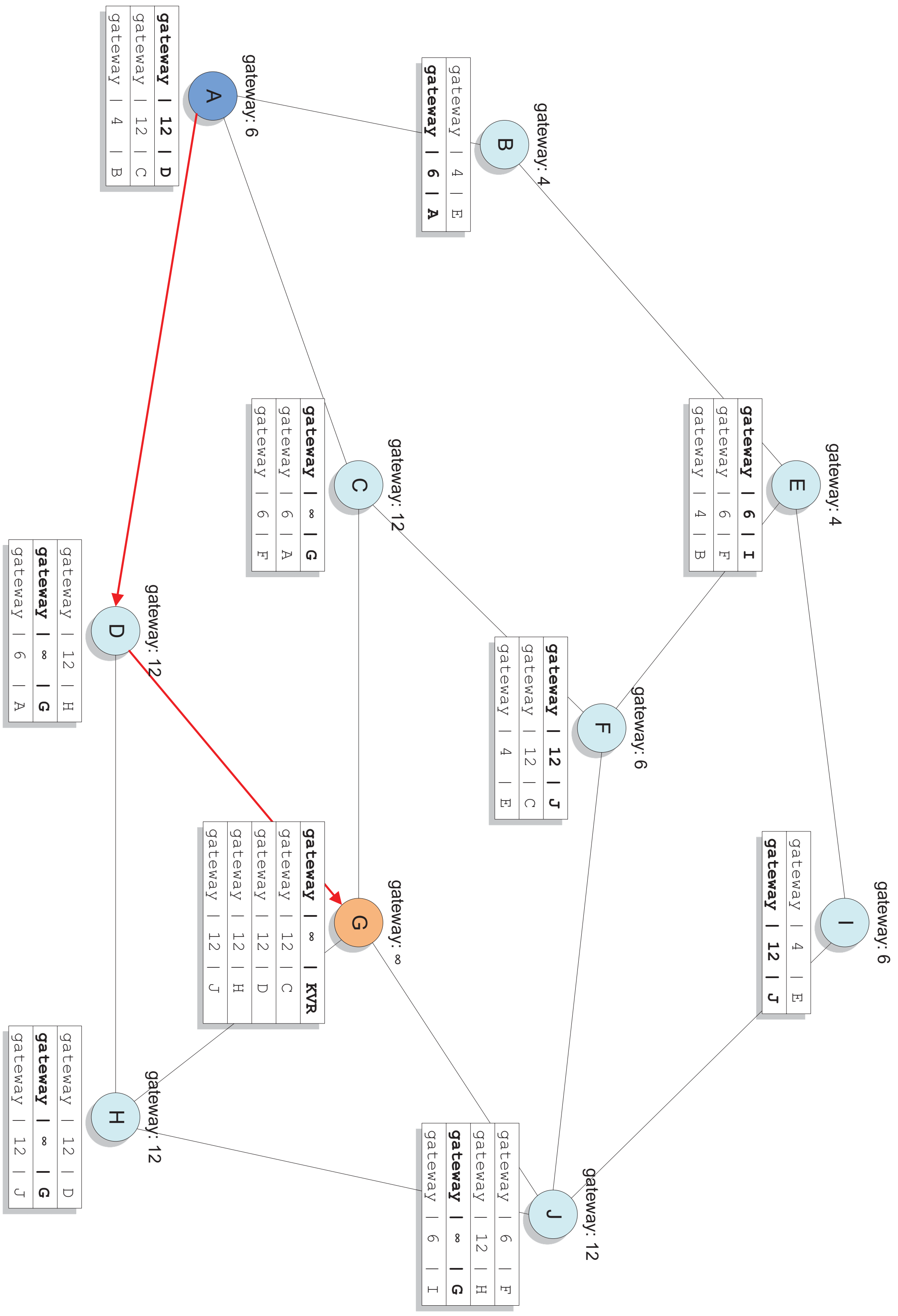


Figure 5.27: Network overview for field assembly validation

5.3.3 Multiple Service Instances

In this scenario, two service instances of the same service type are started on different nodes. The interesting points are the superstition of both field quotients and the novel path from a client application to the service instance which the compartment discovered. As one can see in figure 5.28, the additional service instance is started on node I with a CoS of ”120”. The resulting field potentials for all nodes are shown in table 5.2:

Node	*Potential for service type ”gateway”*
A	6 + 40 = 46
B	4 + 60 = 64
C	12 + 40 = 52
D	12 + 40 = 52
E	4 + 120 = 124
F	6 + 60 = 66
G	infinite
H	12 + 60 = 72
I	infinite
J	12 + 120 = 132

Table 5.2: Computed field potentials after publishing two service instances

After publishing the ”gateways” on node G and I, a client application is started on node A which tries to discover a ”gateway”. There is no need to wait a long time for the complete field assembly and routing information exchange because these processes are triggered by the publishing event. By reason of the route entries as a result of the routing process one can observe the *SUBSCRIBE* message from the client application traveling first to node B, then to node E and in the end to its final, novel destination, node I. On node I, one is able to see on the output of the sample service brick:

```
--> subscribe message received

fbr_serv: message-id: 53617892
fbr_serv: successful register_callback
fbr_serv: anaL2_resolve: Resolve Success
fbr_serv: response message sent to forwarding brick
```

Apparently, the client application discovered the novel service instance successfully. On the output of the sample client brick on node A one can observe:

```
--> response message received

fbr_clie: message-id: 53617892
fbr_clie: message: Hi! I received your subscribe message!
fbr_clie: data message sent
```

Obviously, the client application received a *RESPONSE* message back from the service instance and sends out again some data. Thus, on the sample service brick on node I one can see:

```
--> data message received

fbr_serv: message-id: 53617892
fbr_serv: message: Hi! Now, I know you!
```

In conclusion, the superstition of two field quotients upon publishing two service instances of the same type on node G and on node I was successful on all nodes. Moreover, a client application effectually discovered a better service instance concerning the CoS, hence node A and node I exchanged the message types as expected.

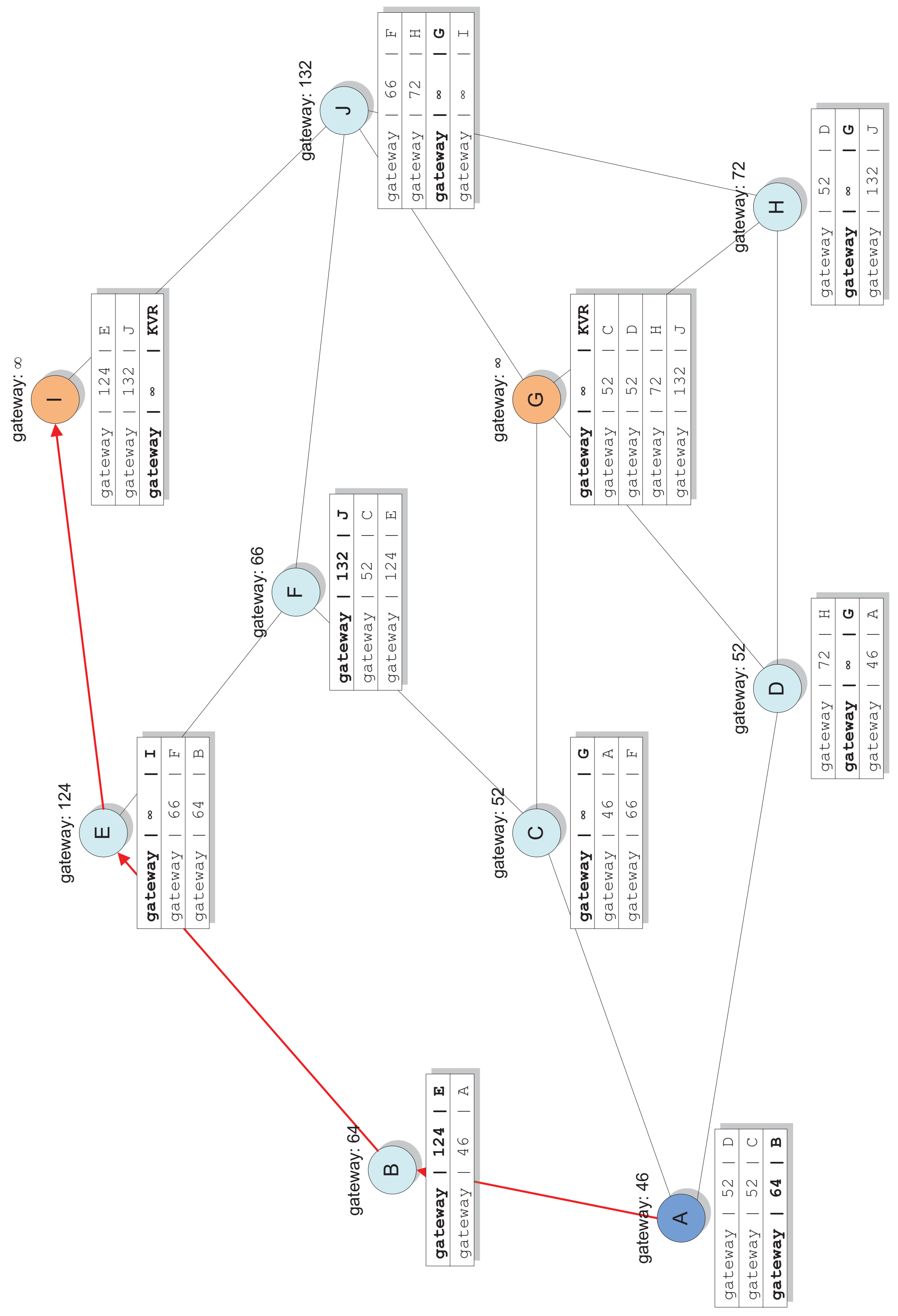


Figure 5.28: Network overview for multiple service instances validation

5.3.4 Multiple Fields

In this scenario, an additional service instance of a different type is started. The interesting point is whether multiple field overlays as a result of multiple service types can exist simultaneously. As one can see in figure 5.29, the other service instance is started on node J with service type ”printer” and a CoS of ”12”. The resulting field potentials for all nodes are shown in table 5.3:

Node	*Potential for service type ”gateway”*	*Potential for service type ”printer”*
A	46	4
B	64	4
C	52	6
D	52	6
E	124	6
F	66	12
G	infinite	12
H	72	12
I	infinite	12
J	132	infinite

Table 5.3: Computed field potentials after publishing two different service types

After publishing a ”printer” on node J, a client application is started on node A which tries to discover the ”printer”. There is no need to wait a long time for the complete field assembly and routing information exchange because these processes are triggered by the publishing event. By reason of the route entries as a result of the routing process one can observe the *SUBSCRIBE* message from the client application traveling first to node D, then to node H and in the end to its final destination, node J. On node J, one is able to see on the output of the sample service brick:

```
--> subscribe message received

fbr_serv: message-id: 73991406
fbr_serv: successful register_callback
fbr_serv: anaL2_resolve: Resolve Success
fbr_serv: response message sent to forwarding brick
```

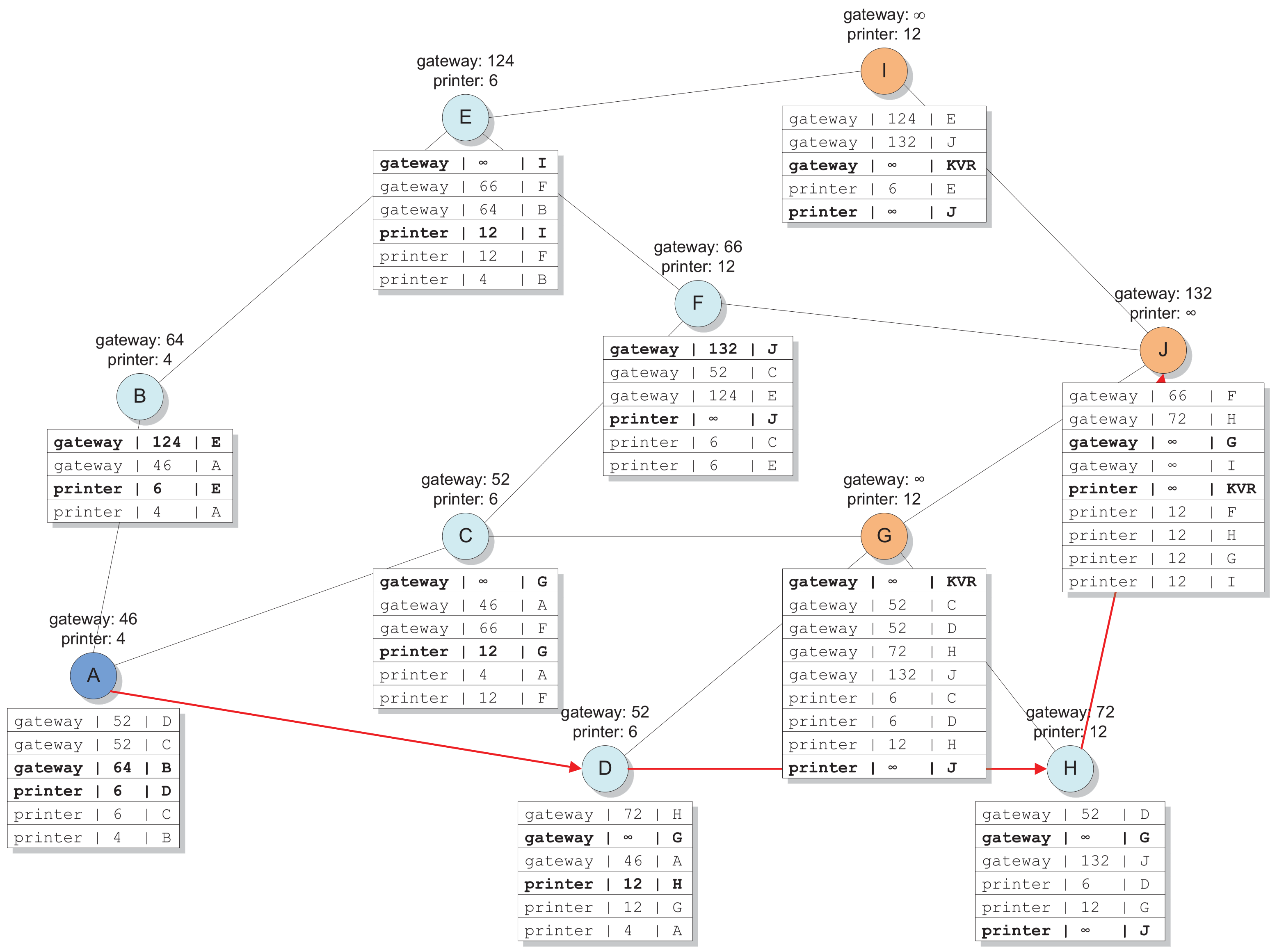
Apparently, the client application discovered the service instance successfully. On the output of the sample client brick on node A one can observe:

```
--> response message received

fbr_clie: message-id: 73991406
fbr_clie: message: Hi! I received your subscribe message!
fbr_clie: data message sent
```

Obviously, the client application received a *RESPONSE* message back from the service instance and sends out again some data. Thus, on the sample service brick on node J, one can see:

```
--> data message received

fbr_serv: message-id: 73991406
fbr_serv: message: Hi! Now, I know you!
```

In conclusion, the field assembly process upon publishing service instances of several service types was successful on all nodes. Multiple field overlays can coexist in the same compartment without any difficulty. Furthermore, a client application effectually discovered the service instance, hence node A and node J exchanged the message types as expected.



gateway	52	D
gateway	52	C
gateway	**64**	**B**
printer	**6**	**D**
printer	6	C
printer	4	B

gateway	**124**	**E**
gateway	46	A
printer	**6**	**E**
printer	4	A

gateway	**∞**	**G**
gateway	46	A
gateway	66	F
printer	**12**	**G**
printer	4	A
printer	12	F

gateway	72	H
gateway	**∞**	**G**
gateway	46	A
printer	**12**	**H**
printer	12	G
printer	4	A

gateway	**∞**	**I**
gateway	66	F
gateway	64	B
printer	**12**	**I**
printer	12	F
printer	4	B

gateway	**132**	**J**
gateway	52	C
gateway	124	E
printer	**∞**	**J**
printer	6	C
printer	6	E

gateway	**∞**	**KVR**
gateway	52	C
gateway	52	D
gateway	72	H
gateway	132	J
printer	6	C
printer	6	D
printer	12	H
printer	**∞**	**J**

gateway	52	D
gateway	**∞**	**G**
gateway	132	J
printer	6	D
printer	12	G
printer	**∞**	**J**

gateway	124	E
gateway	132	J
gateway	**∞**	**KVR**
printer	6	E
printer	**∞**	**J**

gateway	66	F
gateway	72	H
gateway	**∞**	**G**
gateway	∞	I
printer	**∞**	**KVR**
printer	12	F
printer	12	H
printer	12	G
printer	12	I

Figure 5.29: Network overview for multiple fields validation

5.3.5 Node Failure

In this scenario, an arbitrary node is killed simulative. The interesting point is whether the routing process is able to find new routes from a client to the service instances. As one can see in figure 5.30, the node killed is node D. Therefore, the path discovered from node A to node J, regarding the service type "printer", is interrupted. The resulting field potentials for all nodes are shown in table 5.4:

Node	*Potential for service type "gateway"*	*Potential for service type "printer"*
A	46	4
B	64	4
C	52	6
D	killed	killed
E	124	6
F	66	12
G	infinite	12
H	72	12
I	infinite	12
J	132	infinite

Table 5.4: Computed field potentials after killing one node

After killing node D, a client application is started on node A which tries to discover a "printer". Depending on the parameters given (in section 5.2) it takes some time until all routing tables are updated with the correct routing information, because the inoperative entries of the routing tables have to die out. In this case, regarding the default parameters, it took between two and three minutes. Afterwards, one can observe the *SUBSCRIBE* message from the client application traveling now first to node C, then to node G and in the end to its final destination, node J. On node J, one is able to see on the output of the sample service brick:

```
--> subscribe message received

fbr_serv: message-id: 36942092
fbr_serv: successful register_callback
fbr_serv: anaL2_resolve: Resolve Success
fbr_serv: response message sent to forwarding brick
```

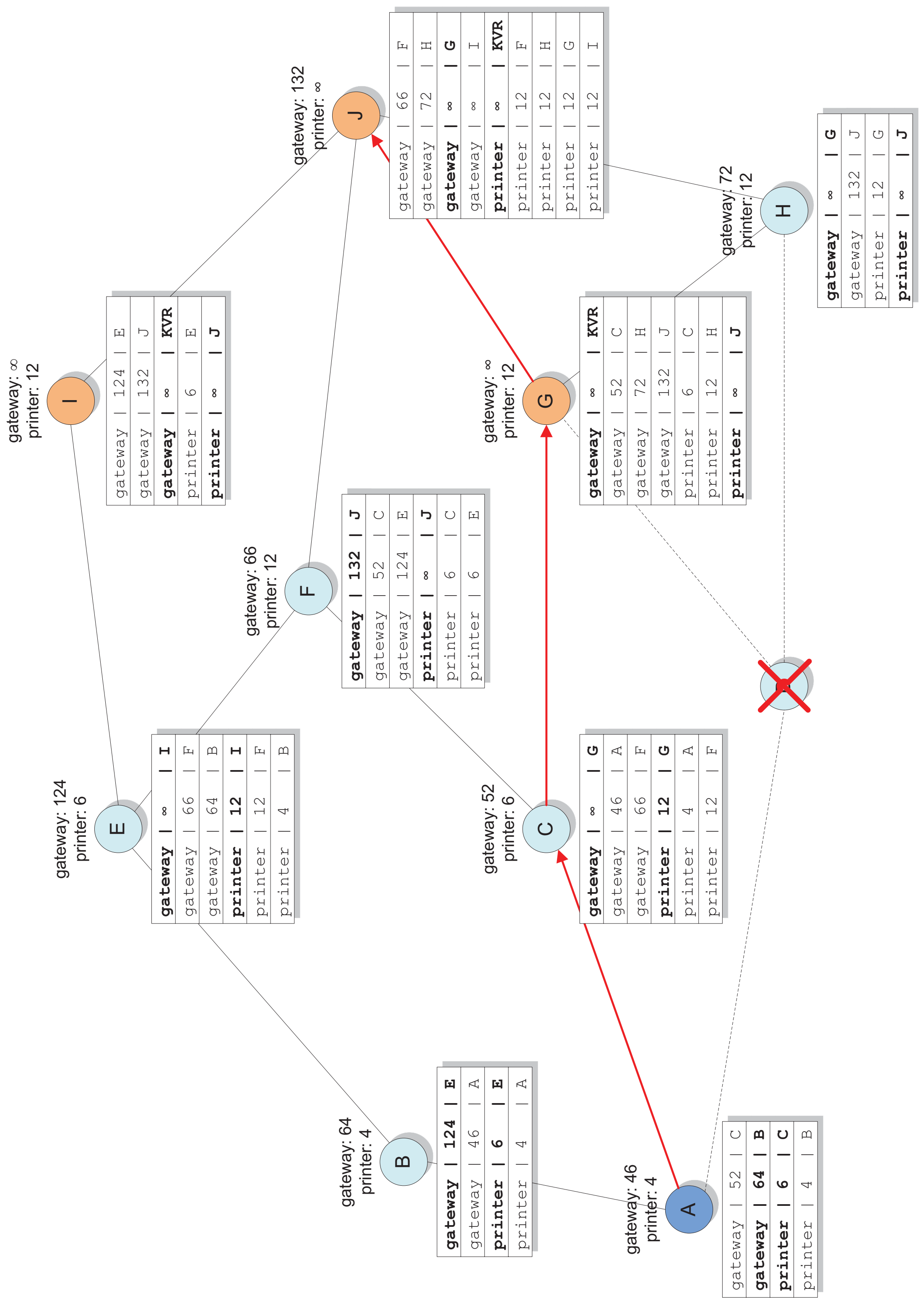
Apparently, the client application discovered the service instance successfully. On the output of the sample client brick on node A one can observe:

```
--> response message received

fbr_clie: message-id: 36942092
fbr_clie: message: Hi! I received your subscribe message!
fbr_clie: data message sent
```

Obviously, the client application received a *RESPONSE* message back from the service instance and sends out again some data. Thus, on the sample service brick on node J, one can see:

```
--> data message received

fbr_serv: message-id: 36942092
fbr_serv: message: Hi! Now, I know you!
```

In conclusion, node A has found a new path to the service type ”printer” successfully. Node A and node J exchanged the message types as expected.



gateway	124	E
gateway	132	J
gateway	**∞**	**KVR**
printer	6	E
printer	**∞**	**J**

gateway	66	F
gateway	72	H
gateway	**∞**	**G**
gateway	∞	I
printer	**∞**	**KVR**
printer	12	F
printer	12	H
printer	12	G
printer	12	I

gateway	**∞**	**G**
gateway	132	J
printer	12	G
printer	**∞**	**J**

gateway	**∞**	**KVR**
gateway	52	C
gateway	72	H
gateway	132	J
printer	6	C
printer	12	H
printer	**∞**	**J**

gateway	**132**	**J**
gateway	52	C
gateway	124	E
printer	**∞**	**J**
printer	6	C
printer	6	E

gateway	**∞**	**I**
gateway	66	F
gateway	64	B
printer	**12**	**I**
printer	12	F
printer	4	B

gateway	**∞**	**G**
gateway	46	A
gateway	66	F
printer	**12**	**G**
printer	4	A
printer	12	F

gateway	**124**	**E**
gateway	46	A
printer	**6**	**E**
printer	4	A

gateway	52	C
gateway	**64**	**B**
printer	**6**	**C**
printer	4	B

Figure 5.30: Network overview for node failure validation

5.4 Summary

The ultimate goal of field-based service discovery is publishing a specific service type network-wide in order to provide the opportunity for client applications of discovering a specific service type by address-agnostic subscribe messages. This is achieved by assembling a scalar field overlay for the network, depending on the capacity of service (CoS) of the service instances as well as the hop count as distance metrics. Therefore, a client application, showing its intent to discover a service type is always guided to an optimal service instance regarding the service load and the network conditions.

Field-based service discovery as exploring novel and cutting-edge network protocol was designed and implemented in the ANA Playground successfully. Thanks to the modular decomposition, the design discloses exceeding flexibility whereas each protocol piece can be exchanged and extended easily. The design of field-based service discovery fits excellently in the ANA world. The architecture of one node participating in the compartment contains five individual and specialized bricks, separating and modeling the fundamental functionality of the protocol:

- Field computing brick: The field computing brick is the heart of the architecture dealing with the field assembly for a specific service type.
- Routing table brick: The routing table brick is responsible to select neighbour nodes with the highest potential.
- Dissemination brick: The dissemination brick exchanges information between nodes and strongly separates between field assembly and routing information.
- Forwarding brick: The forwarding brick cares about forwarding payload after a client succeeded in discovering a service instance by sending a subscribe message for this specific service type.
- Forwarding table brick: The forwarding table brick holds, as the name implies, the forwarding table supporting the forwarding process which is done along the steepest gradient of the field.

Together, the described bricks build the functional block of field-based service discovery. The communication, hence the interfaces between the bricks and the nodes, are elaborately defined in order to provide the aspired flexibility. Node internal as well as node external communication is encoded as XRP messages acting as dynamic containers with a well-defined format but dynamic content.

Several crucial validation scenarios exemplify impressively the implementation of field-based service discovery for ANA. The scenario of field assembly shows the process of distributing field assembly information upon publishing a service instance. The scenario of multiple service instances pictures out the superstition of different field quotients. Furthermore, multiple fields coexisting simultaneously are also possible. Finally, the node failure scenario which leads to novel routes in the nodes, tops off the complete and proper implementation of field-based service discovery for ANA.

Chapter 6

Summary and Further Work

The last chapter summarizes the contributions and objectives achieved for ANA by this Master Thesis. The conclusion intends to highlight again the novel features developed. Moreover, the chapter gives an outline about possible enhancements and further work conceivable.

6.1 Summary

The ANA project builds an autonomic network based on a clean slate approach. The ultimate goal is to develop a novel network architecture that enables flexible, dynamic and full autonomic formation of networks nodes as well as whole networks. Therefore, the network stack is not fixed as in the Internet but it is dynamically built depending on the network needs. This flexibility is achieved by defining a MINMEX containing the required functionality to run ANA on the one hand and the ANA Playground containing actual network functionality, protocols and applications on the other hand.

This Master Thesis is situated in the ANA project, more exactly in the ANA Playground. The achieved goal is to develop new protocols and applications to be designed, implemented and tested in the existing ANA prototype. Therefore, the following improvements have been developed and validated successfully:

- The design and implementation of the Internet Protocol (IP) for ANA as first use case for a complex network protocol.
- The design and implementation of the Routing Information Protocol (RIP) for ANA as first opportunity to perform routing inside an ANA network.
- The design and implementation of field-based service discovery for ANA as exploring novel and cutting-edge protocol, combining field-based routing and address-agnostic service discovery.

IP in ANA exhibits some variances compared to the standard IP. Nevertheless, the most important features such as IP header, encapsulation, checksum computation, forwarding and the addressing scheme are supported accurately, extending the features of ANA increasingly. Because of the IP compartment it is now possible to communicate over disparate Ethernet segments whereas before, only messages inside the same Ethernet segment was a possible communication. In addition, RIP offers the full functionality of the standard RIP.

Field-based service discovery as brand-new protocol provides publishing service types network-wide in order that clients are able to discover the service types by address-agnostic subscribe messages. This is achieved by assembling a scalar field overlay for the network, depending on the capacity of service (CoS) of the service instances.

Thanks to the modular design of all protocols, the decomposition discloses exceeding flexibility whereas each protocol piece can be exchanged and extended easily. The architecture of one node participating in the compartments contains individual and specialized bricks, separating the fundamental functionality of the protocols. In the end, several crucial validation scenarios exemplify impressively the implementation of the novel features enriching the ANA Playground.

Besides these main topics, a large variety of other task were performed, reaching from code debugging to giving a presentation on an ANA coding workshop in Liège (Belgium). In conclusion, the development of new protocols and applications for ANA resulted with IP and RIP as well as field-based service discovery in two major and functioning features for ANA and it was a pleasure to participate in the ANA development team.

6.2 Further Work

ANA is an ongoing project and the ANA software is far from being finished. The most interesting and desirable enhancements arised during the development of this Master Thesis are listed below:

- As already described, IP in ANA does not offer the full functionality of the standard IP. Therefore it could be necessary to develop additional bricks providing for instance fragmentation, IP options or ICMP. Thanks to the modular design, the development of additional bricks for the functional block of IP is possible at any time.
- An important enhancement would be a functional block implementing a reliable transport layer protocol. This is necessary to be able to re-send lost IP datagrams. For example SAFT (Store And Forward Transport (SAFT) [52]) is well suited for wireless environments.
- The ANA project enforces the concept of dynamic protocol stacks. Therefore it could be interesting to try running IP without Ethernet. This is possible by binding the IP encapsulation brick directly to the vlink brick instead of the Ethernet brick. Obviously, one has to think about new ideas replacing today's ARP process.
- For now, the configuration of an ANA IP node is done manually by the configuration brick. To be convenient to ANAs concept of autonomic networks, new ideas are imaginable like Zeroconf [41] in order to provide full autonomic address configuration.
- Field-based service discovery runs on top of Ethernet but there are again other protocol stacks conceivable. It is possible to run field-based service discovery for example on top of IP. One only has to change a few source code lines in the dissemination and forwarding brick to perform resolve processes for neighbour nodes in the IP compartment instead of the Ethernet compartment. Moreover, it should be possible to chose the desired compartment with auxiliary arguments at runtime because the IP and Ethernet compartment handle resolve processes in the same way.
- The feedback information channel for response messages in the field-based service discovery compartment is implemented by storing the path of subscribe messages in the nodes. As already mentioned there are a lot of other ideas possible to offer a feedback channel such as providing network addresses of other protocols (for example IP) or computing additional field overlays for clients.
- Another interesting point for further work is the information exchange between nodes in the field-based service discovery compartment. It should be possible to fill the dynamic containers described in this Master Thesis with RIP routing information instead

of field-based routing information. Therefore, it could be possible to chose routing mechanisms dynamically.

- In addition to the hardware abstraction layer, an ANA abstraction layer would be useful. It should allow legacy applications to run over ANA. Therewith one could evaluate the benefits of the ANA network architecture in comparison with the current Internet.

Last but not least, the ANA Playground provides endless free space for developing further protocols and applications imaginable.

Appendix A

How-to: Start IP and RIP

A.1 Compilation

You can get and compile the ANA source code as follows [40]:

1. Get the source code [51]:

   ```
   svn checkout https://subversion.cs.unibas.ch/repos/ana/
   ```

   You will need a username and a password. These can be obtained from Christophe Jelger from the University of Basel.

2. Switch to the trunk directory and compile the program:

   ```
   cd ana/ana-core/trunk/
   make user
   ```

   Note that for now the IP and RIP bricks only work in user space.

A.2 Loading the MINMEX and the Bricks

1. MINMEX:

   ```
   ./bin/minmex -b 1000
   ```

   The MINMEX control gate will be shown on the output which es needed to attach the bricks ("-b" is used to extend the maximal message size which is necessary for RIP advertisements).

2. vlink: Virtual link brick [4]

   ```
   ./bin/vlink -n MINMEX_CONTROL_GATE
   ```

   You need to have root privileges.

3. Configure the vlink:

   ```
   ./bin/vlconfig create 1 1
   ./bin/vlconfig add_if vlink1 eth0
   ./bin/vlconfig up vlink1
   ```

   For more detailed instructions to configure the vlink, refer to [4].

4. dgrmEthBrick: Ethernet brick [4]

   ```
   ./bin/dgrmEthBrick -n MINMEX_CONTROL_GATE
   ```

   You need to repeat step 3 and 4 in order to install multiple interfaces. For each vlink configured, one dgrmEthBrick has to be started. Therefore, you need to change the names of the additional Ethernet bricks with an auxiliary argument:

   ```
   ./bin/dgrmEthBrick -n MINMEX_CONTROL_GATE -a eth02
   ```

5. ip_enc: Encapsulation brick

   ```
   ./bin/ip_enc -n MINMEX_CONTROL_GATE
   ```

6. ip_sum: Checksum computing brick

   ```
   ./bin/ip_sum -n MINMEX_CONTROL_GATE
   ```

7. ip_fwd: Forwarding table brick

   ```
   ./bin/ip_fwd -n MINMEX_CONTROL_GATE
   ```

8. ip_cfg: Configuration brick

   ```
   ./bin/ipconfig -n MINMEX_CONTROL_GATE -...
   ```

   Please refer to section 4.2 for instructions how to use the configuration brick.

9. rip: RIP brick

   ```
   ./bin/rip -n MINMEX_CONTROL_GATE
   ```

10. (ip_usr: Sample user brick, refer to section 4.2)

For a more detailed discussion on how to load, configure and troubleshoot the MINMEX and the bricks, please refer to [2].

A.3 Start with Shell Script

- There is also the possibility to run an ANA node with IP by dint of a shell script (single interface):

  ```
  ./C/bricks/ip/start_anaip.sh VLINK_ID INTERFACE IP_ADDRESS DEFAULT_GATEWAY
  ```

  Where:

  – `VLINK_ID`: The virtual link ID you want to configure for the `INTERFACE`.
  – `INTERFACE`: The interface you want to use, e.g. eth0.
  – `IP_ADDRESS`: The IP address for your ANA node (a 255.255.255.0 subnet mask will be assumed).
  – `DEFAULT_GATEWAY`: The IP address of the default gateway.

- After having started ANA and the IP compartment successfully you can use screen [46] to observe the outputs of the bricks:

  – Show a specific output (”reattach a screen”):

    ```
    screen -r {ip_enc, ip_sum, ip_fwd}
    ```

  – Ctrl+A + Ctrl+D to close the output (”detach a screen”)

Appendix B

How-to: Start Field-Based Service Discovery

B.1 Compilation

You can get and compile the ANA source code as follows [40]:

1. Get the source code [51]:

   ```
   svn checkout https://subversion.cs.unibas.ch/repos/ana/
   ```

   You will need a username and a password. These can be obtained from Christophe Jelger from the University of Basel.

2. Switch to the trunk directory and compile the program:

   ```
   cd ana/ana-core/trunk/
   make user
   ```

   Note that for now the IP and RIP bricks only work in user space.

B.2 Loading the MINMEX and the Bricks

1. MINMEX:

   ```
   ./bin/minmex
   ```

   The MINMEX control gate will be shown on the output which es needed to attach the bricks.

2. vlink: Virtual link brick [4]

   ```
   ./bin/vlink -n MINMEX_CONTROL_GATE
   ```

   You need to have root privileges.

3. Configure the vlink:

   ```
   ./bin/vlconfig create 1 1
   ./bin/vlconfig add_if vlink1 eth0
   ./bin/vlconfig up vlink1
   ```

For more detailed instructions to configure the vlink, refer to [4].

4. dgrmEthBrick: Ethernet brick [4]

   ```
   ./bin/dgrmEthBrick -n MINMEX_CONTROL_GATE
   ```

   You need to repeat step 3 and 4 in order to install multiple interfaces. For each vlink configured, one dgrmEthBrick has to be started. Therefore, you need to change the names of the additional Ethernet bricks with an auxiliary argument:

   ```
   ./bin/dgrmEthBrick -n MINMEX_CONTROL_GATE -a eth02
   ```

5. fbr_diss: Dissemination brick

   ```
   ./bin/fbr_diss -n MINMEX_CONTROL_GATE
   ```

6. fbr_forw: Forwarding brick

   ```
   ./bin/fbr_forw -n MINMEX_CONTROL_GATE
   ```

7. fbr_ftab: Forwarding table brick

   ```
   ./bin/fbr_ftab -n MINMEX_CONTROL_GATE
   ```

8. fbr_rtab: Routing table brick

   ```
   ./bin/fbr_rtab -n MINMEX_CONTROL_GATE
   ```

9. fbr_potf: Field computing brick

   ```
   ./bin/fbr_potf -n MINMEX_CONTROL_GATE
   ```

10. (fbr_serv: Sample service brick, refer to section 4.2)
11. (fbr_clie: Sample client brick, refer to section 4.2)

For a more detailed discussion on how to load, configure and troubleshoot the MINMEX and the bricks, please refer to [2].

B.3 Start with Shell Script

- There is also the possibility to run with field-based service discovery by dint of a shell script:

  ```
  ./C/bricks/fbr/start_fbr.sh
  ```

  Note that the MINMEX, vlink and Ethernet bricks have to be running already.

- After having started the shell script successfully you can use screen [46] to observe the outputs of the bricks:

  - Show a specific output (”reattach a screen”):

    ```
    screen -r {fbr_diss, fbr_forw, fbr_ftab, fbr_rtab, fbr_potf}
    ```

  - Ctrl+A + Ctrl+D to close the output (”detach a screen”)

Appendix C

Doxygen Code Documentation

C.1 IP and RIP

Data Structure Documentation

devAddr Struct Reference

struct **devAddr**

Data Fields

- anaLabel_t **dev**
- char * **addr**
- char * **mask**

Detailed Description

struct **devAddr**

A data structure to store IP addresses of interfaces.

Field Documentation

anaLabel_t devAddr::dev

Interface

char* devAddr::addr

IP address

char* devAddr::mask

Netmask

The documentation for this struct was generated from the following file:

- **ip.h**

fullTarget Struct Reference

struct **fullTarget**

Data Fields

- char * **address**
- anaLabel_t **label**
- char * **destDescrip**
- char * **source**
- uint8_t **type**
- char * **srcDescrip**

Detailed Description

struct **fullTarget**

A data structure to store the properties for communication targets.

Field Documentation

char* fullTarget::address

IP address

anaLabel_t fullTarget::label

Outgoing interface

char* fullTarget::destDescrip

IP user application

char* fullTarget::source

IP source to use

uint8_t fullTarget::type

CMD or next header

char* fullTarget::srcDescrip

Source description

The documentation for this struct was generated from the following file:

- **ip.h**

resolveRequestList Struct Reference

struct **resolveRequestList**

Data Fields

- char **resId** [RES_ID_SIZE]
- anaLabel_t **replyTo**
- void * **context**
- int **contextLen**
- int **timerId**
- void * **target**
- int **targetLen**
- char **chanType**
- void * **description**
- int **descLen**

Detailed Description

struct **resolveRequestList**

A data structure to store pending resolve requests.

Field Documentation

char resolveRequestList::resId[RES_ID_SIZE]

Resolve ID

anaLabel_t resolveRequestList::replyTo

IDP to send the response to

void* resolveRequestList::context

Context

int resolveRequestList::contextLen

Length of the context

int resolveRequestList::timerId

Timer ID

void* resolveRequestList::target

Target

int resolveRequestList::targetLen

Length of the target

char resolveRequestList::chanType

Channel type

void* resolveRequestList::description

Source description

int resolveRequestList::descLen

Length of the source description

The documentation for this struct was generated from the following file:

- **ip.h**

File Documentation

ip.h File Reference

Data Structures

- struct **fullTarget**
  *struct **fullTarget***
- struct **devAddr**
  *struct **devAddr***
- struct **resolveRequestList**
  *struct **resolveRequestList***

Functions

- uint32_t AGENTCLASSMEMBER **dottedQuadToHex** (char *dotted)
  A function to transform an IP address from dotted quad to hexadecimal notation.
- char *AGENTCLASSMEMBER **hexToDottedQuad** (uint32_t hex)
  A function to transform an IP address from hexadecimal to dotted quad notation.
- char *AGENTCLASSMEMBER **num_mask** (uint32_t mask)
  A function to transform a netmask to short notation (e.g. "\24").

Detailed Description

IP header file with necessary data structures, defines (documented in the report) and functions.

Function Documentation

uint32_t AGENTCLASSMEMBER dottedQuadToHex (char * *dotted*)

A function to transform an IP address from dotted quad to hexadecimal notation.

Parameters:

dotted IP address in dotted quad notation

Returns:

IP address in hexadecimal notation

See also:

hexToDottedQuad()

char* AGENTCLASSMEMBER hexToDottedQuad (uint32_t *hex*)

A function to transform an IP address from hexadecimal to dotted quad notation.

Parameters:

hex IP address in hexadecimal notation

Returns:

IP address in dotted quad notation

See also:

dottedQuadToHex()

char* AGENTCLASSMEMBER num_mask (uint32_t *mask*)

A function to transform a netmask to short notation (e.g. "\24").

Parameters:

mask Netmask in hexadecimal notation

Returns:

Short notation as string

ip_enc.c File Reference

Functions

- uint8_t AGENTCLASSMEMBER **generateNextHeader** (void)
  A function to generate a next header code for the IP next header field.

Detailed Description

IP encapsulation brick.

Function Documentation

uint8_t AGENTCLASSMEMBER generateNextHeader (void)

A function to generate a next header code for the IP next header field.

Returns:

New random next header code

C.2 Field-Based Service Discovery

Data Structure Documentation

ids Struct Reference

struct **ids**

Data Fields

- char * **type**
- char * **id**
- int **seq**
- int **param1**
- int **param2**
- int **age**

Detailed Description

struct **ids**

A data structure to store service IDs and according sequence numbers of all service instances.

Field Documentation

char* ids::type

Service type

char* ids::id

Service ID

int ids::seq

Ongoing sequence number

int ids::param1

Parameter 1, e.g. capacity

int ids::param2

Parameter 2, e.g. hop count

int ids::age

Age for garbage collection issues

The documentation for this struct was generated from the following file:

- **fbr.h**

mids Struct Reference

struct **mids**

Data Fields

- char * **type**
- char * **id**
- int **cown**
- int **sown**
- anaLabel_t **clabel**
- anaLabel_t **slabel**
- char * **requester**

Detailed Description

struct **mids**

A data structure to store the message IDs of all messages.

Field Documentation

char* mids::type

Service type

char* mids::id

Message ID

int mids::cown

Client application is node local

int mids::sown

Service instance is node local

anaLabel_t mids::clabel

Next IDP towards client application

anaLabel_t mids::slabel

Next IDP towards service instance

char* mids::requester

Origin requester

The documentation for this struct was generated from the following file:

- **fbr.h**

neighbour Struct Reference

struct **neighbour**

Data Fields

- char * **id**
- anaLabel_t **label**
- int **age**

Detailed Description

struct **neighbour**

A data structure to store all **neighbour** nodes.

Field Documentation

char* neighbour::id

Unique ID for **neighbour**, e.g. MAC address

anaLabel_t neighbour::label

Outgoing interface

int neighbour::age

Age for garbage collection issues

The documentation for this struct was generated from the following file:

- **fbr.h**

pots Struct Reference

struct **pots**

Data Fields

- char * **type**
- char * **field**
- float **potential**
- int **age**

Detailed Description

struct **pots**

A data structure to store field potentials of all service instances.

Field Documentation

char* pots::type

Service type

char* pots::field

Field type

float pots::potential

Field potential

int pots::age

Age for garbage collection issues

The documentation for this struct was generated from the following file:

- **fbr.h**

services Struct Reference

struct **services**

Data Fields

- char * **type**
- char * **id**
- int **seq**
- char * **field**
- int **param1**
- int **param2**
- int **ttl**
- int **age**

Detailed Description

struct **services**

A data structure to store the properties of node local service instances.

Field Documentation

char* services::type

Service type

char* services::id

Service ID

int services::seq

Ongoing sequence number for advertisements

char* services::field

Field type

int services::param1

Parameter 1, e.g. capacity

int services::param2

Parameter 2, e.g. hop count

int services::ttl

Time to live for advertisements

int services::age

Age for garbage collection issues

The documentation for this struct was generated from the following file:

- **fbr.h**

tab Struct Reference

struct **tab**

Data Fields

- char * **type**
- char * **potential**
- int **own**
- anaLabel_t **label**
- int **age**

Detailed Description

struct **tab**

A data structure to store the routing table.

Field Documentation

char* tab::type

Service type

char* tab::potential

Field potential

int tab::own

Indicates type of next hop (local or remote node)

anaLabel_t tab::label

Next hop

int tab::age

Age for garbage collection issues

The documentation for this struct was generated from the following file:

- **fbr.h**

File Documentation

fbr.h File Reference

Data Structures

- struct **services**
  *struct **services***
- struct **ids**
  *struct **ids***
- struct **pots**
  *struct **pots***
- struct **tab**
  *struct **tab***
- struct **neighbour**
  *struct **neighbour***
- struct **mids**
  *struct **mids***

Functions

- int **randomsnr** (int a, int e)
  A function to generate a random number between a lower and upper bound.

Detailed Description

FBSD header file with necessary data structures, defines (documented in the report) and functions.

Function Documentation

int randomsnr (int *a*, int *e*)

A function to generate a random number between a lower and upper bound.

Parameters:

a Lower bound
e Upper bound

Returns:

Random number

Bibliography

[1] The ANA Project, Autonomic Network Architecture, http://www.ana-project.org (March 2008)

[2] Ariane Keller, Development of the ANA Core Software, Master Thesis, ftp://ftp.tik.ee.ethz.ch/pub/students/2007-So/MA-2007-38.pdf (March 2008)

[3] ANA Blueprint, Sixth Framework Program, Project Number: FP6-IST-27489, Deliverable D1.4/5/6 Version 1.0, Christophe Jelger et al

[4] ANA Core Documentation, https://subversion.cs.unibas.ch/repos/ana/ana-core/trunk/doc/anacore-doc.pdf (March 2008)

[5] State of the Art, Sixth Framework Programm, Project Number: FP6-IST-27489, Deliverable 1.1, Christophe Jelger et al

[6] R. D. Pethia, Computer Security, Testimony Before the Committee on Government Reform Subcommittee on Government Management, Information and Technology, March 9, 2000

[7] M. Handley and A. Greenhalgh, Steps Towards a DoS-Resistant Internet Architecture, Proceedings of ACM SIGCOMM Workshop on Future Directions in Network Architecture, 2004

[8] A. Nakao, L. Peterson and A. Bavier, A Routing Underlay for Overlay Networks, Proceedings 2003 Conference on Applications, Technologies, Architectures and Protocols for Computer Communications, 2003

[9] L. Peterson, S. Shenker and J. Turner, Overcoming the Internet Impasse through Virtualization, Proceedings of 3rd Workshop on Hot Topics in Network (HotNets-III), 2004

[10] Active Technologies, Vorlesungsslides von B. Plattner, SS 2007

[11] Situated and Autonomic Communications (SAC), http://cordis.europa.eu/ist/fet/comms.htm

[12] Larry L. Peterson and Bruce S. Davie, Computer Networks: A Systems Approach, 3rd Edition

[13] Christian Benvenuti, Understanding Linux Network Internals, O'Reilly

[14] Klaus Wehrle, Frank Pählke, Hartmut Ritter, Daniel Müller and Marc Bechler, The Linux Networking Architecture, Design and Implementation of Network Protocols in the Linux Kernel

[15] Open Systems Interconnection Reference Model (OSI), ISO Standard 7498-1:1994

[16] International Organization for Standardization (ISO), http://www.iso.org/iso/home.htm

[17] Internet Protocol, RFC 791

[18] Internet Protocol Version 6 (IPv6), RFC 2460

[19] Transmission Control Protocol, RFC 793

[20] User Datagram Protocol, RFC 768

[21] Requirements for Internet Hosts - Communication Layers, RFC 1122

[22] Assigned Numbers, RFC 1340

[23] Internet Control Message Protocol (ICMP), RFC 792

[24] Routing Information Protocol, RFC 1058

[25] A Border Gateway Protocol 4 (BGP-4), RFC 4271

[26] Open Shortest Path First Protocol (OSPF), RFC 2740

[27] Vincent Lenders, Martin May and Bernhard Plattner, Service Discovery in Mobile Ad Hoc Networks: A Field Theoretic Approach, Pervasive and Mobile Computing 1 (2005) 343-370

[28] Anindya Basu, Alvin Lin and Sharad Ramanathan, Routing Using Potentials: A Dynamic Traffic-Aware Routing Algorithm, in Proceedings of the ACM Annual Conference of the Special Interest Group on Data Communication, SIGCOMM 2003, Karlsruhe, Germany, August 2003

[29] Vincent Lenders, Field-Based Routing and its Application to Wireless Ad Hoc Networks, Diss., ETHZ, Nr. 16681, 2006

[30] Nam T. Nguyen, An-I Andy Wang, Peter Reiher and Geoff Kuenning, Electric-Field-Based Routing: A Reliable Framework for Routing in MANETs, Mobile Computing and Communications Review, Volume 8, Number 1

[31] Ulas C. Kozat and Leandros Tassiulas, Network Layer Support for Service Discovery in Mobile Ad Hoc Networks, in Proceedings of the IEEE INFOCOM, San Francisco, USA, April 2003

[32] Vincent Lenders, Martin May and Bernhard Plattner, Density-Based vs. Proximity-Based Anycast Routing for Mobile Networks

[33] The ANA Project, Deliverable 2.8 Version 1.0, Service Discovery and Routing Schemes for intra- and inter-Compartment Service Provisioning, http://www.ana-project.org/deliverables/2007/ana-d2.8-final.pdf (March 2008)

[34] Chenxi Wang, Antonio Carzaniga, David Evans and Alexander L. Wolf, Security Issues and Requirements for Internet-Scale Publish-Subscribe Systems, http://www.cs.virginia.edu/~evans/pubs/hicss.pdf (March 2008)

[35] Yongqiang Huang and Hector GarciaMolina, Publish/Subscribe in a Mobile Environment, Department of Computer Science, Stanford, http://infolab.stanford.edu/~yhuang/papers/mobpubsub.pdf (March 2008)

[36] The ANA Project, ANA Wiki, WP1, Task 1.4, Lookup API, https://www.ana-project.org/wiki/workpackages/wp1/task-1-4/lookup-api (March 2008)

[37] Address Resolution Protocol, RFC 826

[38] ETH Zürich, Computer Engineering and Networks Laboratory, 8092 Zurich, http://www.tik.ee.ethz.ch

[39] TIK Testbed, http://tiknet.ee.ethz.ch/doku.php (March 2008)

[40] ANA Code Repository, https://subversion.cs.unibas.ch/repos/ana

[41] Zero Configuration Networking (Zeroconf), http://www.zeroconf.org

[42] Traditional IP Network Address Translator (Traditional NAT), RFC 3022

[43] Session Initiation Protocol (SIP), RFC 3261

[44] Ethereal / Wireshark, http://www.wireshark.org/

[45] Valgrind, http://valgrind.org/

[46] Screen, http://www.gnu.org/software/screen/

[47] Doxygen Source Code Documentation Generator Tool, http://www.stack.nl/~dimitri/doxygen/index.html

[48] Richard Bellman, On a Routing Problem, in Quarterly of Applied Mathematics, 16(1), pp. 87-90, 1958

[49] E. W. Dijkstra, A Note on Two Problems in Connexion With Graphs, in Numerische Mathematik 1, pp. 269–271, 1959

[50] Gnutella, http://www.gnu.org/philosophy/gnutella.html (March 2008)

[51] Subversion, http://subversion.tigris.org/

[52] SAFT, Store And Forward Transport, Simon Heimlicher, Master Thesis , ftp://ftp.tik.ee.ethz.ch/pub/students/2004-2005-Wi/MA-2005-08.pdf (March 2008)