ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Autonomic Identifier Allocation for ANA (AIA)

Mathias Fischer

Semester Thesis
Winter Semester 2008
January 7, 2009

SA-2008-23

Supervisor: Prof. Dr. B. Plattner
Advisors: Ariane Keller, Theus Hossmann, Dr. Martin May
Computer Engineering and Networks Laboratory, ETH Zurich

Abstract

In new networking architectures autonomy is very important – novel networks should be able to organise and configure themselves autonomously. One of the most important parts of configuring applications and protocols is identifier allocation, because for most applications having a unique identifier is crucial, in order to be able to communicate with each other. In this semester thesis we designed and implemented an autonomous and distributed system that is able to provide unique identifiers to all kinds of applications and protocols in the Autonomic Networking Architecture (ANA) framework. It generates random identifiers based on a regular expression provided by the requesting application, and checks them for uniqueness in a specified scope. The use of 'Autonomic Identifier Allocation' simplifies configuration and therefore the set up of networks and brings an important portion of autonomy to the ANA project.

Contents

1 Introduction

The ANA (Autonomic Network Architecture) [1] project[1] aims at exploring novel ways of organising and using networks beyond legacy Internet technology. ANA is designed to provide a networking system that is more flexible than the fixed network stack of today's Internet. The ANA project states that a new architecture should be designed in a way that its functionality is scalable – an architecture should allow both different functionality and different ways of implementing a given functionality.
One of ANA's key properties is autonomy. According to the project, novel networks should be able to organise themselves. This minimises the effort of administrators to set up complicated networks. A very important part of configuring a network and its applications and protocols is identifier allocation. For most applications it is crucial to have unique identifiers, in order to be able to communicate with each other. IP needs unique IP addresses, Ethernet needs unique MAC addresses, even a chat application needs unique user names in order to work properly.
Even though ANA aims to be autonomous and despite the importance of unique identifiers, ANA lacks of a system that is able to automatically allocate identifiers. Identifiers, for example IP addresses, need to be configured manually. This task can become very complicated as a network grows and hosts join and leave the network.
In this semester thesis we designed and implemented an autonomous and distributed system that is able to provide its identifier allocation functionality to all kinds of applications and protocols. It generates different types of identifiers and checks them for uniqueness in a specified scope. This system is called *Autonomic Identifier Allocation* or short *AIA*.
The remainder of this thesis is organized as follows: In Section 2 we will write about the most important related projects. Section 3 will explain important ANA abstractions and structures that are important to understand AIA. The design and implementation of *Autonomic Identifier Allocation* will be described in Section 4. In Section 5 we will show some validation and evaluation tests. In Section 6 finally, we will describe where this work can be further improved and what points would be interesting to have a deeper look into.

[1] See Section 3 for a detailed description of the ANA project.

2 Related Work

In this section, the most important projects dealing with autonomic name and address allocations will be described. First, we will describe Zeroconf [2], probably the most famous project in this area of which Bonjour [3] is a specific implementation. The Unique Identifier Allocation Protocol [4] is a system that checks different types of identifiers, provided by an application, for their uniqueness. Finally, IP Address Configuration Algorithm for Zeroconf Mobile Muti-hop Ad-Hoc Networks [5], describes a distributed algorithm for Ad-Hoc networks that is based on agents that maintain a list of all assigned IP addresses.

2.1 Zeroconf

Zeroconf [2] is a working group which aims at developing a technique to connect computers and devices in a local network without the need of manual configuration. It consists of three main parts:

- automatic allocation of IP addresses without DHCP
- automatically resolve and distribute host names
- automatic service discovery

The first part of Zeroconf is described in the Dynamic Configuration of IPv4 Link-Local Addresses [6] which is based on ARP (Address Resolution Protocol) [7]. According to this document, a host basically chooses an address out of the 169.254/16 network (without the reserved 256 first and 256 last addresses) and claims that this is its new address. This is done by sending ARP probes – broadcast ARP messages with source address set to 0.0.0.0 and destination set to the newly chosen IP address. If another host already uses this address it responds with an ARP reply message what causes the claiming host to give up and choose another address. Once an address claim was successful, that means for a given time no ARP messages for the same IP have been received, ARP announcements are broadcasted over the network to make sure all hosts know the newly chosen address. ARP announcement messages are similar to ARP probes but with source and destination address set to the new IP address.
If a host receives an ARP message whose source IP address is equal to its own address there must be a conflict. Conflicts can occur during the process of finding an IP address because Zeroconf first choses a random IP and only then checks whether this address is still available. On the other hand, conflicts can also occur after all host already had assigned an address, due to the connection of previously disjoint networks.
After a successful claim, a host doesn't give up its new address straight away if a conflict occurs – it tries to defend the address by sending an ARP announcement message. To avoid deadlocks where two or more hosts try to defend the same IP address, a host gives up its IP address if it sees multiple conflicts during a certain time period.

2.2 Bonjour

Bonjour [3] (formerly known as Rendezvous) is a Zeroconf implementation by Apple. The address allocation part basically works similar to [6]. Bonjour is

only needed if the host fails to receive an address from a DHCP server. On Apple OS-X, if Bonjour address allocation is used, not one but ten addresses are claimed in series and the first of them working is taken. Even if Bonjour sets a link-local address, DHCP is tried every five minutes. Since modern operating systems are able to detect whether an Ethernet interface is connected, Bonjour address allocation can be triggered by plugging a cable.

2.3 Unique Identifier Allocation Protocol

The Unique Identifier Allocation Protocol [4] is a by now expired internet draft. It proposes a protocol which offers a service to applications which assures that a requested name is only allocated if it's not already in use by another instance of the application in the network. It supports multiple types of identifiers (e.g. network addresses or host names). The identifier, however, is chosen by the application itself. The Unique Identifier Allocation Protocol checks the availability of the identifier and returns whether it's still available. An availability check is done by claiming an identifier and waiting for hosts denying this claim. If none of the hosts deny the claim during a specified period of time, the identifier is still available.

2.4 Zeroconf Mobile Muti-Hop Ad-Hoc Networks

In [5], the authors propose an agent-based addressing algorithm for ad-hoc networks. Each node can either be bound (having an address) unbound (having no address) or an agent. A node in the unbound state waits for an agent to send a Verify-Packet with the network's full address list. A Verify-Packet is basically a periodic request from the agent to all nodes to verify their address. If no address list is received for a certain period of time, the node decides to switch to the agent state. Otherwise the node sends an Address-Request to the agent. Nodes that are already bound to an address have to confirm their address after each received Verify-Packet. The agent therefore gets aware of all used addresses and is able to provide addresses to requesting nodes.

3 ANA

Before describing the Autonomic Identifier Allocation system in Section 4, we will provide some explanations of ANA, its functionality and important abstractions used.

3.1 Goal

The ANA (Autonomic Network Architecture) [1] project aims at exploring and implementing new ways of networking. Its ultimate goal is the flexible and autonomous formation of network nodes as well as whole networks. In today's internet architecture, IP is the most important protocol – at the same time it is the main technologic bottleneck – there are many protocols that run on IP and many technologies IP can run on, but all devices that are connected to the internet need to communicate through the Internet Protocol. The ANA project claims that a new architecture should be designed in a way that its functionality is scalable – both horizontally (different functionality) and vertically (different ways of integrating abundant functionality).
ANA is a project participated by numerous universities and research institutes. Its objective is to provide an architectural framework which allows the implementation of the full range of network technologies, such as small local area networks, mobile ad-hoc networks or even global scale networks like the Internet. An important part of ANA is autonomy – it is the intention of the project to facilitate the self-* features of autonomic networking such as self-management, self-organisation and self-configuration.

3.2 Architecture

ANA mainly consists of two parts, the MINMEX – a core which has to be provided by every ANA node, and the playground where the actual network functionality is implemented. The playground consists of multiple so called bricks which provide some functionality to the network, e.g. encryption, compression or connection management. Multiple bricks can be conceptually grouped to functional blocks – which provide a service to the network. For example the functional block IP consists of multiple bricks for forwarding, routing or checksum calculation. All bricks communicate via the MIMNEX core using the ANA compartment API.
Functional blocks which are part of the same node or share the same network protocol are in the same *compartment*. A network compartment can be seen as an abstraction similar to a subnet or a virtual network. A node compartment is basically the set of all functional blocks (and their interconnection) of a single node.
The compartment abstraction allows the decomposition of communication networks into smaller and manageable units. In addition a compartment hides its internal implementation, it only provides some specified functionality to functional blocks.
In order to allow for dynamic rebinding of the information flow between functional blocks, Information Dispatch Points (IDPs) have been introduced. Communication between functional blocks happens always via IDPs. The Information Dispatch Table (IDT) stores the association between bricks and IDPs.

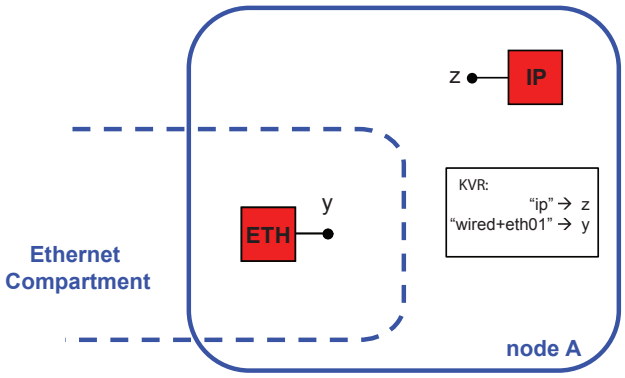

Figure 1: ANA abstractions at the example of a node with ETH and IP functional blocks, Ethernet segment and the KVR.

Compartments are identified by keywords which describe their service. The Key Value Repository (KVR) stores the association between these keywords and the corresponding IDPs.
The abstraction of a communication service provided by a compartment is called information channel. Information channels only exist inside the bounds of a compartment. Figure 1 shows a node with an Ethernet compartment, two functional blocks and the KVR storing the associations of IDPs and keywords.

3.3 The Compartment API

Compartments require some kind of registration function allowing bricks to become a member of the compartment. A similar function is needed to leave the compartment. Compartments that use identifiers to name or address their members need to manage the identifier space themselves. A compartment which uses identifiers also needs to provide a function which is able to resolve an identifier and obtain and return the information how to to communicate with this entity. Further compartments typically include routing functionality in order to find a communication path between any information source and destination within the compartment.
In ANA a context describes the region in which a service is available, this can be restricted to a certain subnet, to the local node (context `"."`) or to the maximum reachable context (context `"*"`).
The following primitives implement the registration and the resolution functionality, respectively:

publish The primitive $\texttt{IDP}_\texttt{p}$ `= publish(`$\texttt{IDP}_\texttt{c}$`,context,service,timeout)` requests from a compartment (identified by $\texttt{IDP}_\texttt{c}$) that a certain service (`service`) becomes reachable via the compartment in a certain context (`context`), it returns an IDP ($\texttt{IDP}_\texttt{p}$) by which the service can be accessed. The `timeout` parameter specifies the maximum time allowed for this action.

resolve The primitive `IDP_i = resolve(IDP_C,context,service,timeout)` is asking a compartment (again identified by `IDP_C`) for an information channel to a certain service (`service`) in a given context (`context`), it returns the IDP (`IDP_i`) of the information channel requested. The `timeout` parameter specifies the maximum time allowed for this action, after that time `resolve` returns without having found the service.

3.4 Terminology

According to the ANA Blueprint [8] names, addresses and identifiers can be distinguished as follows:

Identifier An identifier is a finite sequence of symbols of a given alphabet, that is used to identify a certain entity of a set, an identifier therefore can be either a name or an address.

Name A name is a globally unique identifier used for recognition of a given entity. Usually names aim at being easily recognisable by humans, for example a chat user name.

Address An address is an identifier that is not only used for identification but also for routing and forwarding purposes. An IP address for example does not only identify a host but also contains information how a route can be found to it.

The abstractions and terms introduced in this section are very important for the further understanding of the functionality of AIA, which is described in the following section.

4 Design and Implementation

This section shows the design and implementation of AIA. One of the basic questions about the design of AIA is how the allocation of an identifier is triggered, what will be described in the first subsection. Once AIA has been triggered, an identifier has to be generated - how this is done is shown in the next subsection. The centrepiece of this thesis - the allocation process - is explained in Subsection 4.3. We will also show how AIA is extended such that addresses can even be allocated in disjoint compartments. Further, we will have a look at the problem that the same identifier could be allocated to multiple instances of an application. After pointing out the most important functions and threads in AIA and how the program control flows between them, we will explain introduced messages and what they are used for. In the end of this section the API will be described.

4.1 Initiation

Since ANA aims to be scalable in functionality, it doesn't make sense to implement a protocol that only assigns IP addresses within Ethernet Compartments. It should be possible to assign arbitrary identifiers in an arbitrary compartment. This generic property is definitely one of the most important points about the design of an identifier allocation mechanism for ANA.
The mentioned projects in Section 2 follow different paradigms how an address (or more general: identifier) allocation process is initiated. Bonjour [3] for example is initiated when a cable is plugged and no IP address could be found using a DHCP server. UIAP [4] on the other hand leaves the initialization to the client application. In this case, the next question is whether the name allocation process also needs to generate an appropriate identifier or whether this is left to the application.
Since a generic allocation process doesn't know anything about the structure of the address space of the compartments, the identifier should be chosen by the application or protocol that needs an identifier. On the other hand, we do not want too much functionality of the allocation and generation process being placed in the applications themselves.
In order to achieve a generic identifier allocation process, that still doesn't need the applications to generate an identifier we decided to let the application describe its identifier space with a regular expression. Further the application has to define the compartment in which an identifier needs to be unique. And finally it can specify a timeout which is the maximum time AIA should try to find an available identifier before giving up.

4.2 Identifier Generation

As mentioned above, the generation of identifiers is based on regular expressions [9] given by the application. Regular expressions are already used by the *cfinder* [10] brick to identify identifiers belonging to a certain brick. Regular expressions can describe any kind of address or name space – even an address space's structure, like subnetting in the example of IP addresses, can be implemented – since some parts of the IP address can be kept fixed whereas other parts can be described as being variable. It is even possible to provide a list of allowed names

in the form of a regular expression. Some examples for name spaces described by regular expressions:

IPv4 `(([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])\\.){3}`
`([0-9]|[1-9][0-9]|1[0-9]{2}|2[0-4][0-9]|25[0-5])`

Zeroconf address range `169\\.254\\.`
`([1-9]|([1-9][0-9])|(1[0-9][0-9])|(2[0-4][0-9])|25[0-4])\\.`
`([0-9]|([1-9][0-9])|(1[0-9][0-9])|(2[0-4][0-9])|25[0-5])`

MAC address `(([0-9]|[A-F]):){5}([0-9]|[A-F])`

List `(alpha|beta|gamma|delta)`

The actual generation of an identifier is done by calling a Perl script. Perl is very mighty in terms of regular expressions and there is a module that does exactly what's needed for the generation of identifiers based on regular expressions: It provides a function that generates a string that matches a given regular expression. In order to be able to call this function, the following two Perl modules need to be installed:

- `Parse::RandGen::Regexp`
- `YAPE::Regex`

The Perl script is called with the `popen` command which opens a UNIX pipe to a command line program. It is called as follows:

```
regen.pl regex
```

where `regex` is a Perl regular expression, or in other words a string representing a regular expression enclosed by backslashes. For example:

```
regen.pl /10\.0\.0\.[0-9]/
```

4.3 Allocation Process

Once an identifier is requested from an application or a protocol, the identifier allocation process is triggered (See Section 4.8 for a more detailed explanation of the API). The centrepiece of AIA is given by the following steps depicted by Figure 2:

1. AIA randomly generates an identifier based on the regular expression given by the requesting application or protocol (requesting brick).
2. The compartment in which the identifier needs to be unique is called underlying compartment. The underlying compartment is described by a keyword, therefore it's also possible to request an identifier that is unique among multiple compartments connected to the node of the requesting brick. AIA resolves the generated identifier in the underlying compartments. In other words, this means that the compartments are asked to look for a service named with that given identifier in the whole compartment. (See Section 3.3 for a more detailed explanation of the Compartment API).

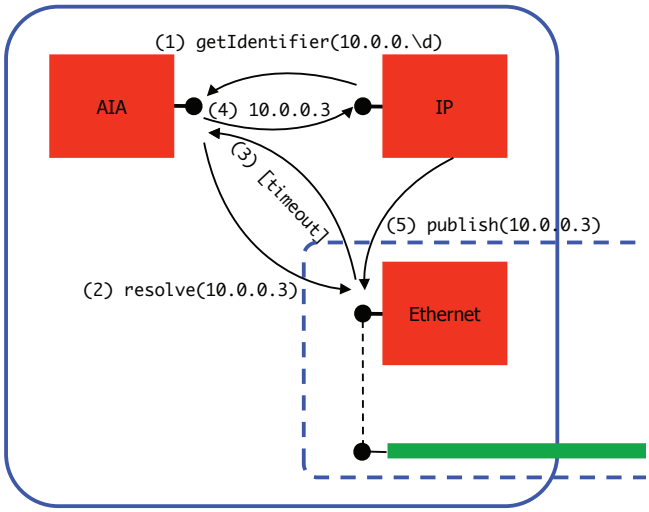

Figure 2: Diagram of the allocation process at the example of IP on an Ethernet segment. First IP requests an identifier based on a regular expression (1). After AIA generates a matching identifier, it tries to resolve it in the Ethernet compartment (2). If the resolve request times out (3), AIA can return the identifier to IP (4), which can publish it in the Ethernet compartment (5).

- If the resolve request is successful, the identifier was found in the compartment. That means that there is already a service using that identifier and it therefore cannot be returned to the application.
  → Go to step 1.
- If the resolve request times out (the timeout for `resolve` is specified by AIA), the compartment is not able to find a service with the given name. Therefore the generated identifier can safely be returned to the requesting application.

Once an unused identifier could be found and returned to the requesting brick, the application can publish a service named by that identifier in one of the underlying compartments.
If AIA was not able to find an available identifier after a certain number of tries or within the time specified by the requesting brick, it returns `NULL` instead of the identifier. A maximum number of tries is specified to prevent AIA from testing the same identifier over and over again.

4.4 Disjoint Compartments

Zeroconf [2] for example only assigns IP addresses in the same Ethernet segment. However, it is undoubtedly useful to be able to have an IP address that is unique among multiple underlying segments. This does particularly make sense if multiple underlying networks implement different technologies between which

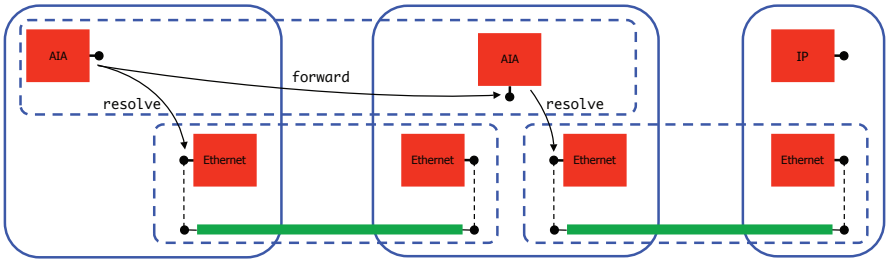

Figure 3: Diagram of forwarding procedure. AIA sends a forward request to all reachable AIA bricks.

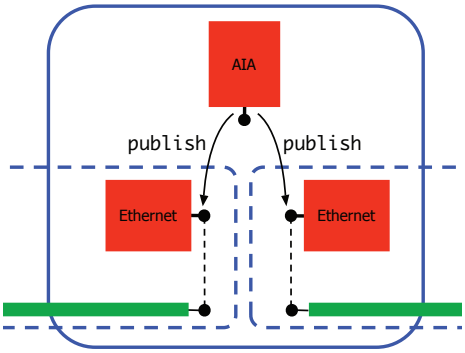

Figure 4: Diagram of AIA publishing itself into multiple underlying compartments. Afterwards this AIA brick is reachable in both compartments.

broadcast messages, and therefore also resolve requests, are not forwarded. It's also useful to assign unique IP addresses among multiple Ethernet segments that are connected to the same node but not yet bridged. If later switching functionality is introduced, the IP addresses are already unique among both segments.

With the above mentioned procedure it's only possible to verify the uniqueness of an identifier in compartments that are available on the local node. In order to be able to verify the uniqueness among compartments that are not directly connected to the local node, the basic procedure of AIA needs to be extended, see Figure 3 for a scheme showing the forwarding procedure.

If an AIA brick publishes itself in all underlying compartments (see Figure 4),it is from then on reachable by all nodes connected by the underlying compartments. In addition to only check local underlying compartments for an identifier's uniqueness, an identifier is forwarded to all reachable AIA bricks where it is checked in all available underlying compartments as well. These AIA bricks forward the request to all reachable AIA bricks again – therefore basically a flooding of AIA forward messages is implemented. In order to distinguish the forward messages belonging to a certain request, they carry a sequence number. And in order to stop the forwarding procedure after a certain time, the messages also carry a hop count. With each hop the timeout value is decreased.

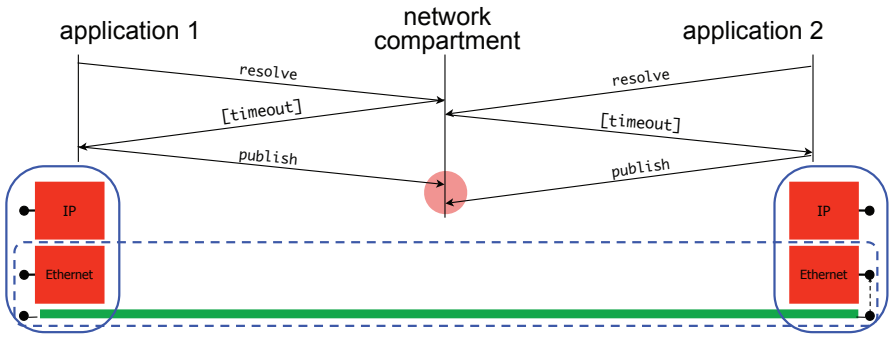

Figure 5: Scheme showing how duplicate identifiers are allowed.

Since the set of underlying compartments is only known after an application requested an identifier, AIA cannot be previously published. This leads to the fact that a node or an AIA brick, respectively can only accept a forwarded request if it was previously used to find an identifier in the same compartment. In other words, in order to achieve unique identifiers throughout all compartments, the interconnecting nodes where compartments are joining, need to request identifiers before other nodes. See Section 6.3 for another possible approach of publishing AIA into compartments.

4.5 Duplicate Identifiers

Since the identifier is in a first phase generated and tested by AIA and then in a second phase published by the application, it is possible that the same identifier gets allocated multiple times. Figure 5 shows how it can happen, that duplicate identifiers are allowed. It shows two different nodes connected by the same underlying compartment. If the two instances of AIA on these two nodes happen to generate the same identifier at the same time, both of them try to resolve this identifier in the underlying compartment. If before, no such identifier existed both resolve request will lead to a timeout and both instances of AIA will allow their requesting applications to publish this identifier, what will lead to a name conflict. Knowing that this problem might occur, we will analyse what factors are important and how the probability of duplicate identifiers can be decreased.
The problem of duplicate identifiers can only occur if two (or more) requests for the same identifier in the same compartment are made at the same time. Therefore three factors are important:

Identifier Space Utilisation Since identifiers are generated randomly, the probability that the same unused identifier is generated twice depends on the usage of the identifier space. If enough identifiers are still available, the probability that the same identifier is generated multiple times is rather small.

Timeout Since the timeout value defines the duration of a resolve request, a

lower timeout decreases the probability of two timeouts happening at the same time. However, a timeout value that is too low, can lead to the problem that a resolve request doesn't find a service even though it would exist.

Request Rate Two resolve requests only happen at the same time, if two identifier requests were made at the same time for the same compartment. Hence, the probability of the problem happening depends on the rate at which identifier requests are made for a given compartment.

For further possibilities to decrease the probability of duplicate identifiers, please refer to Subsection 6.2 in the *Future Work* Section, since this problem is not handled in the current implementation of AIA.

4.6 Implementation of Function Flow

Figure 6 and the following list explain the implemented control and message flow between the components of AIA:

- As mentioned above, a new identifier is triggered by a `REQUEST` from a requesting application.
- The callback function `handleAIARequest`, upon receiving a request calls `findCompartments` which looks up all compartments that match the description given by the application. The local AIA brick is published in all found compartments if this not already done by a previous request. Further in all found underlying compartments, a broadcast AIA IDP is resolved. If this was already done before, a `KEEPALIVE` message is sent to it in order to prevent it from being deleted or to detect a possible deletion. The AIA broadcast IDP is not created permanently, it is therefore deleted by the MINMEX if not used for certain time.
- The callback function then calls the Perl script `regen.pl` that generates an identifier out of the given regular expression.
- This random identifier is passed to the function `sendAIAForward` which launches a `checkerThread` for each available underlying compartment and a `forwarderThread` for the AIA broadcast IDP in each underlying compartment.
- `checkerThread` simply tries to resolve the generated identifier in the underlying compartment and sends the respective `SIGNAL` to `waiterCallback` – this can either be a `ID_USED` or a `TIMEOUT` signal.
- `forwarderThread` sends a `FORWARD` to the AIA broadcast IDP, which reaches all AIA bricks that are published in the respective underlying compartment. Once done, `forwarderThread` also sends a respective `SIGNAL` to `waiterCallback`.
- The function `waiterCallback` blocks until it received a `TIMEOUT` signal from all `checkerThread`s and all `forwarderThread`s or until one of these threads sent an `ID_USED` signal. If the identifier is already used, a new identifier is generated and the procedure is started over again. If all

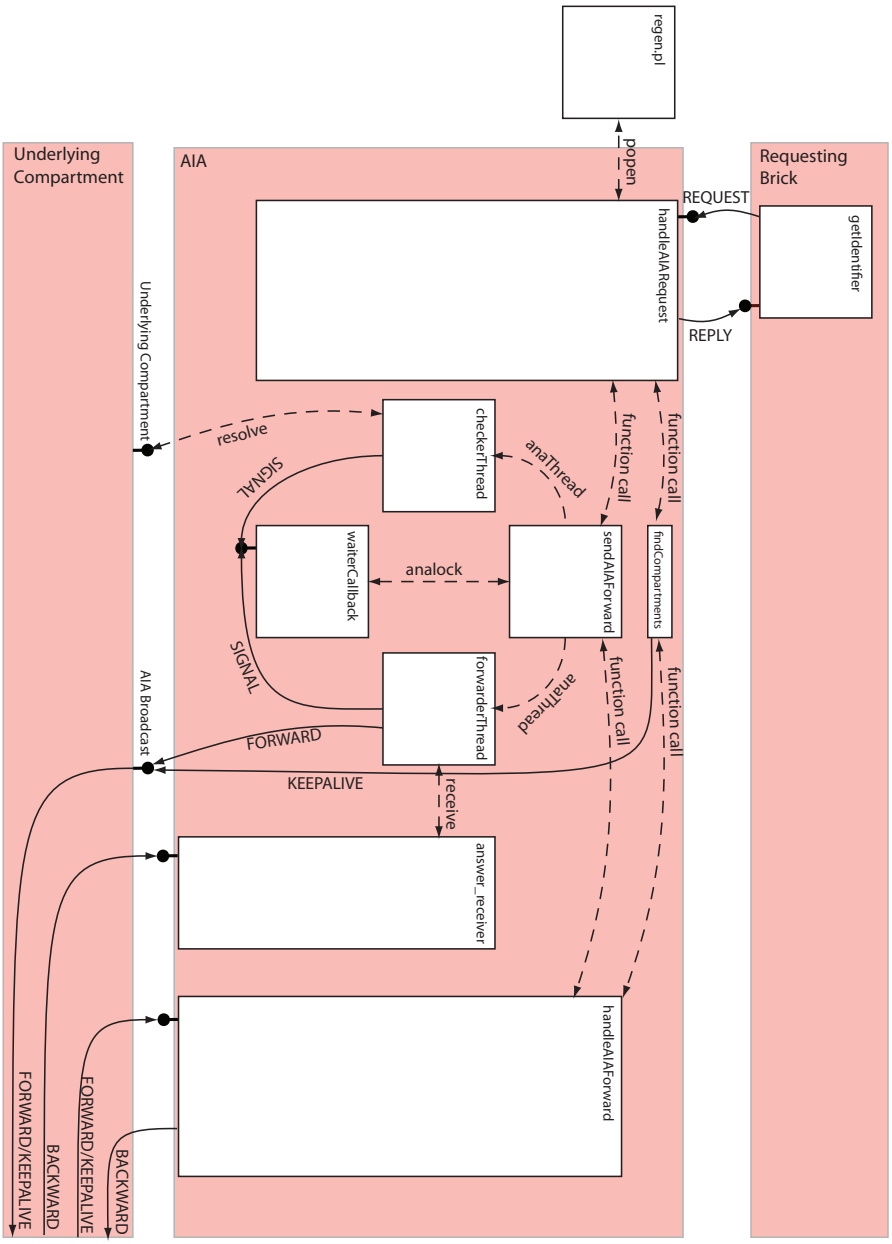

Figure 6: The control and message flow between components of AIA.

threads send a `TIMEOUT` signal, `handleAIARequest` sends a `REPLY` message with the found identifier back to the requesting brick.

- `requestReply` can be used to send a message to an IDP and in the same time request an answer from it, i.e. the function blocks until a reply is received (or the timeout is reached). But since `requestReply` doesn't work as expected to send `FORWARD` messages to a broadcast IDP and receive an answer, the service `answer_receiver` is used by the `forwarderThread` to receive answers from other nodes, it is published (and therefore available) in all underlying compartments.
- If a node receives a `FORWARD` message, `handleAIAForward` calls the function `findCompartments` in order to update the list of underlying compartments, and then calls `sendAIAForward` in order to check the availability of the identifier and forward the the request. If `waiterCallback` receives the signal that the identifier is used, it sends a `BACKWARD` message back to the node that sent the `FORWARD` request (via `answer_receiver`), otherwise it simply doesn't do anything in order to let the request time out.

4.7 XRP Messages

XRP (eXtensible Resolution Protocol) [11] is the standard format for messages between ANA bricks. An XRP message consists of a command (i.e. a message type) and multiple arguments (i.e. values). The following list shows the XRP commands we defined to support the functionality of AIA. Note that the last part of the XRP command name can also be found in Figure 6.

XRP_CMD_AIA_REQUEST is the request that is made out of the requesting application. The function `getIdentifier` which generates a `REQUEST` is provided by `aia.h` to simplify utilisation. A `REQUEST` contains the description of the underlying compartments and the regular expression.

XRP_CMD_AIA_REPLY is a reply to a request from an application. A reply can either carry the identifier string or the message that no identifier could be found.

XRP_CMD_AIA_FORWARD is a forwarding message between two AIA nodes. It consists of a sequence number (in order to distinguish requests), a hop count, the identifier that needs to be checked and the description of the underlying compartments.

XRP_CMD_AIA_BACKWARD is the reply to a forwarding message. A reply to a forward is only sent if the identifier could be found, otherwise the forward request will be let timing out. Therefore a `BACKWARD` messages always consist of the `ID_USED` flag.

XRP_CMD_AIA_SIGNAL is used to inform the `callbackWaiter` function that a `checkerThread` or `forwarderThread` has finished. It's either a `TIMEOUT` or an `ID_USED` signal.

XRP_CMD_AIA_KEEPALIVE is sent to the AIA broadcast IDP in order not to let it disappear because it wasn't used for too long or to detect that it was already deleted. Non-permanently published IDPs are deleted

by the MINMEX if not used for a certain time. A received `KEEPALIVE` message is simply ignored.

4.8 API

AIA provides an API to all other bricks in order to request an identifier. After having imported `aia/aia.h` the API basically only consists of a single function call:

```
char* getIdentifier(char* regex,
                    struct service_s* description,
                    struct timespec* timeout);
```

This function returns a string with the found unique identifier or `NULL` if none could be fount. It takes the following arguments:

- `char* regex` A string with the regular expression that matches all (and only) valid identifiers.
- `struct service_s* description` A `service_s` struct that describes the compartments in which the identifier has to be unique.
- `struct timespec* timeout` Maximum time AIA can try to find an identifier before giving up, without having found one. Note that this timeout doesn't correspond with the timeout parameter of a `resolve` request which is set by AIA.

In the Section A.2 in the appendix we will shortly describe how we adopted the IP configuration and the chat application bricks to use AIA.

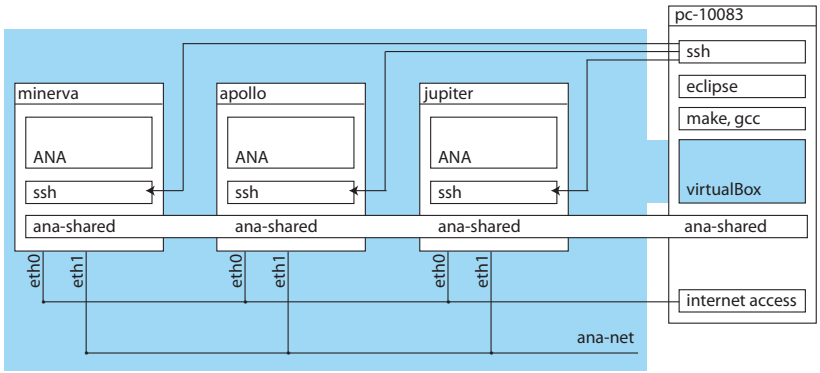

Figure 7: Diagram of the Test Environment

5 Validation and Evaluation

5.1 Developement and Test Environment

To be able to test AIA on a network environment without the need of setting up a network with multiple machines, we decided to use multiple virtual machines running on a single host machine using VirtualBox[2]. For fast and simple compilation and distribution of the new ANA binaries, a folder from the host operating system is shared among all the guest systems. Since all the operating systems are exactly the same (Ubuntu 8.04), we are able to compile the source on the host system and distribute the binaries afterwards. For simpler shell script execution we further installed an ssh server on all guest systems and redirected a certain port to each one of them. See Figure 7 for a detailed view of the test environment.

5.2 Validation

To evaluate correct functionality of AIA, we set up three test scenarios. Figures 8 and 9 show scenarios I and II, respectively. Figure 10 shows scenario III. Figures 8 and 9 show three nodes that are connected by a single compartment in scenario I and two nodes connected to an intermediate node in scenario II, respectively. Figure 10 shows three nodes, where each of them is connected to the two others by a separate compartment. For scenarios I and II, there are three cases each. As seen in the figure, AIA is present in one, two or all nodes, for the other nodes, fixed manually configured identifiers are used.

5.2.1 Scenario I: Single Compartment

Scenario I (Figure 8) is used to test the functionality of AIA in a single compartment. AIA is requested to provide identifiers to chat applications published in an Ethernet segment.

[2]http://www.virtualbox.org/, VirtualBox OSE.

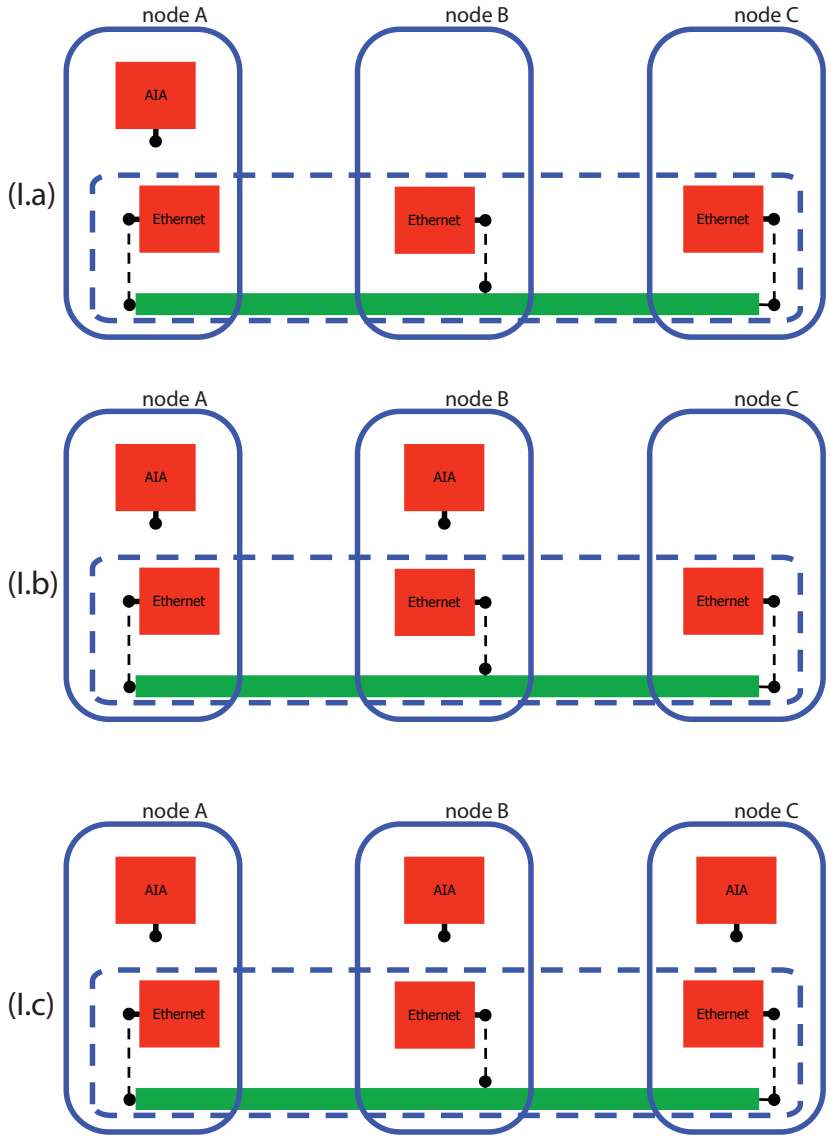

Figure 8: Diagram of testing scenario I, three nodes connected by a single Ethernet compartment. AIA is only used in the nodes with an AIA brick.

First, two of three nodes are configured manually, that means chat names `alpha` and `beta` are given to the chat applications. On the third node, AIA is asked to find an available identifier specified by the simple regular expression `(alpha|beta|gamma)`, assuming correct functionality, only identifier `gamma` is valid. The test is done multiple times and AIA always returns `gamma`, what shows correct functionality for the first case of scenario I.

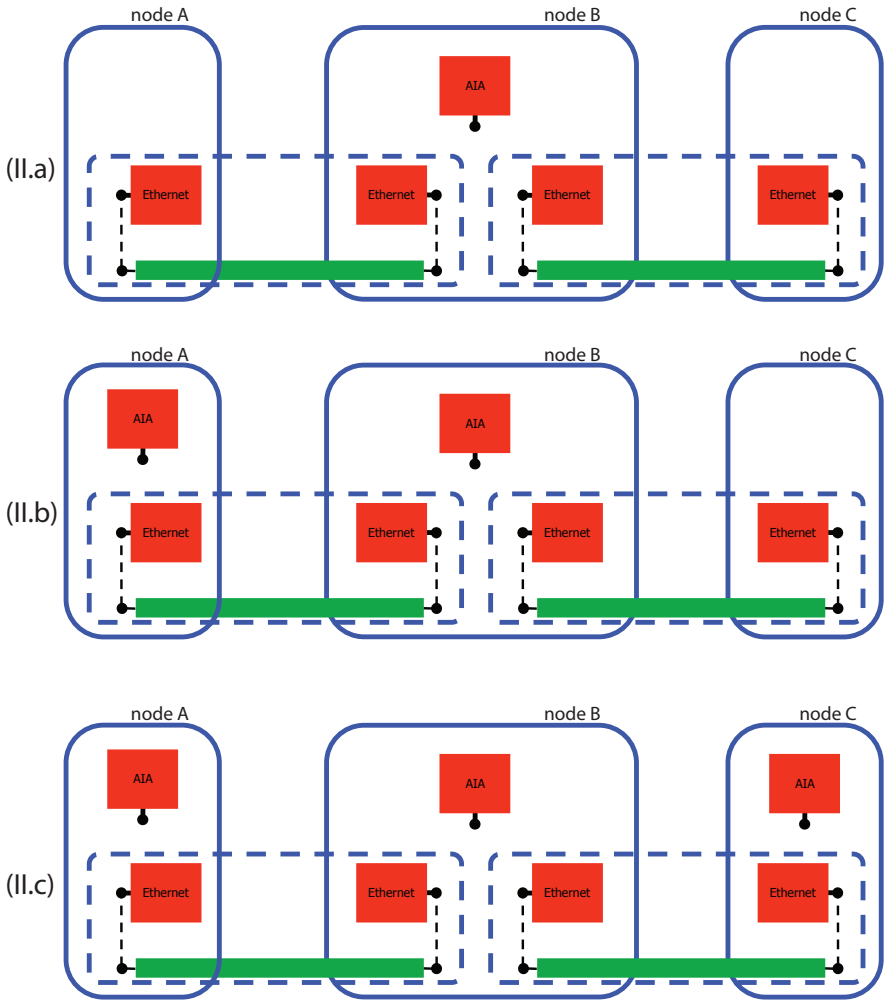

Figure 9: Diagram of testing scenario II, two nodes connected by two Ethernet compartments via an intermediate node. AIA is only used in the nodes with an AIA brick.

A similar test was done for the case where only one node is assigned a fixed chat name - `alpha`. For the same regular expression as above, the remaining two nodes always chose one of the remaining two identifiers `beta` or `gamma`.

The last case for this scenario is a set up of three nodes all using AIA to find a chat name in a single Ethernet compartment. During multiple tests, the three available chat names are always assigned to the three nodes, with none of them being used multiple times. Hence correct functionality could be shown for AIA assigning identifiers in a single compartment.

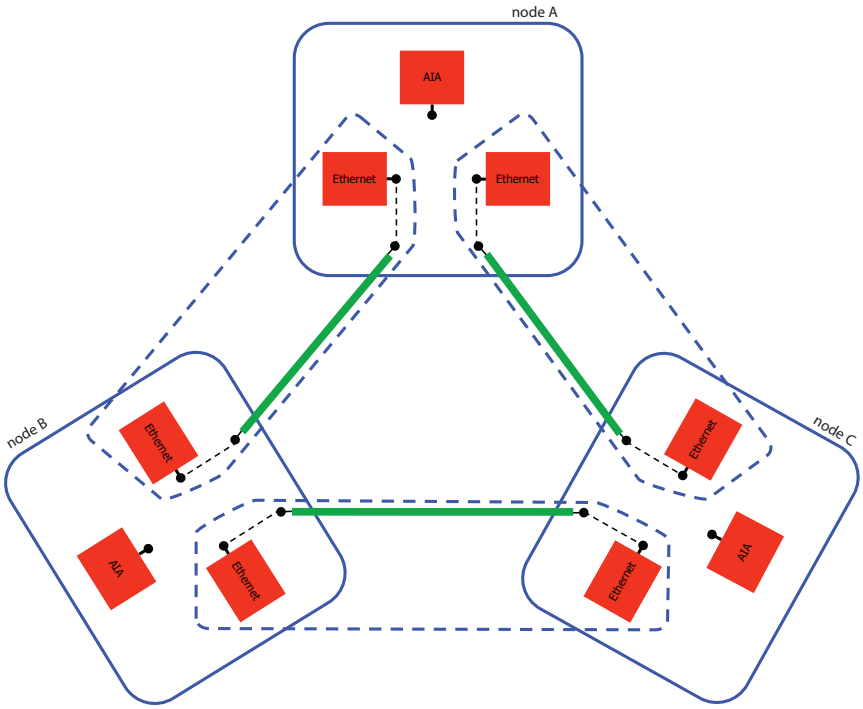

Figure 10: Diagram of testing scenario III, three nodes connected in a circle by Ethernet segments.

5.2.2 Scenario II: One Intermediate Node

With the second validation scenario (Figure 9), correct functionality of checks in multiple compartments and the forwarding mechanism are tested. For the first case only the connecting node (in the middle) will use AIA, the other nodes have fixed user names assigned (again `alpha` and `beta`). AIA successfully finds the only available user name for the middle node.
AIA also works if one or both of the remaining nodes request an identifier. However, for the leftmost and rightmost node (See Figure 9) to have distinct identifiers, the intermediate node needs to request an identifier first, in order to publish itself in the underlying compartment and to be reachable by other AIA bricks. This is already mentioned in Section 4.4, where we also refer to Section 6.3 for an alternative way to publish AIA.

5.2.3 Scenario III: A Circle of Three Compartments

For the last and most complicated scenario IP over Ethernet is used rather than the chat application. Three tests have been done with different regular expressions. First, the ordinary Zeronconf [2] address range was used, where all six interfaces received a distinct IP address with the first generated identifier. Later, the test was done with ten and then six available addresses. In both cases AIA was able to allocate an available IP address to all Ethernet interfaces.

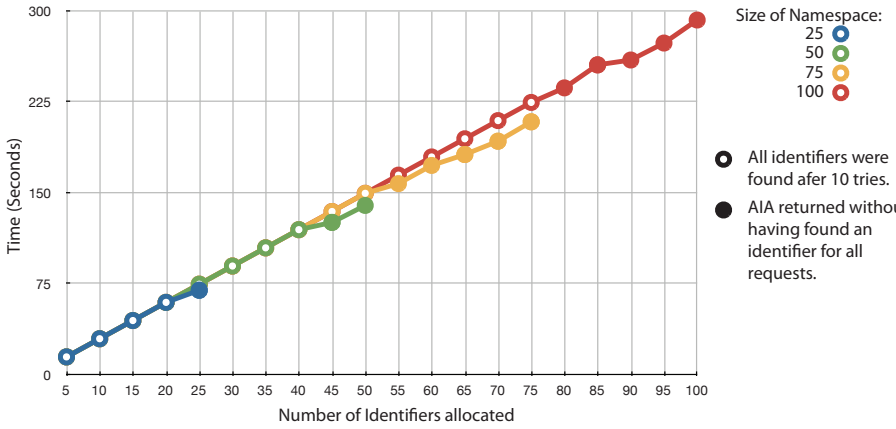

Figure 11: Graph of the first evaluation, time needed to allocate a given number of identifiers.

However in the last case the timeout needed to be increased, since it might take a lot of time to find an IP – especially for the last interface where only one IP is still available.

5.3 Evaluation

The evaluation of AIA was done for two quantities. First the performance – the time to find an available identifier – of AIA was measured. The second evaluation was made for the optimal number of tries. In order to simplify the test setup, all identifiers were allocated for dummy services published on a single node.

For the first evaluation, we measured the time it took for AIA to find a certain number of identifiers depending on different sizes of the identifier space. Figure 11 is the graph of the measurements for four differently sized identifier spaces. The x-axis shows the number of identifiers tried to allocate, the y-axis shows the time it took. The graph shows that the time needed to find a given number of identifiers increases linearly. This is the case because for this measurement all identifiers have been published locally, that means finding an identifier being used is much faster than making sure that it doesn't exist, therefore a second try doesn't take much more time than a single one. Filled circles show the measurement where AIA was not able to allocate the full number of identifiers. In this cases the time is even less than usually since after ten unsuccessful tries, AIA gives up and returns an error. Note that the slope of the lines in Figure 11 correspond to the maximum of the timeout values specified for resolve and forwarding requests. In this case the timeout for a forwarding request is three seconds, AIA is waiting for the timeout to happen before returning the identifier. Therefore the time to check one identifier is bound by this timeout.

The second evaluation deals with the previously mentioned maximum number of tries before AIA gives up finding an identifier. Figure 12 shows the utilisation of the identifier space on the x-axis, where 100% are 100 identifiers. On the y-axis the percentage of identifiers found after a given number of tries is shown. The measurement was done for one, two, five and ten tries. The graph shows that

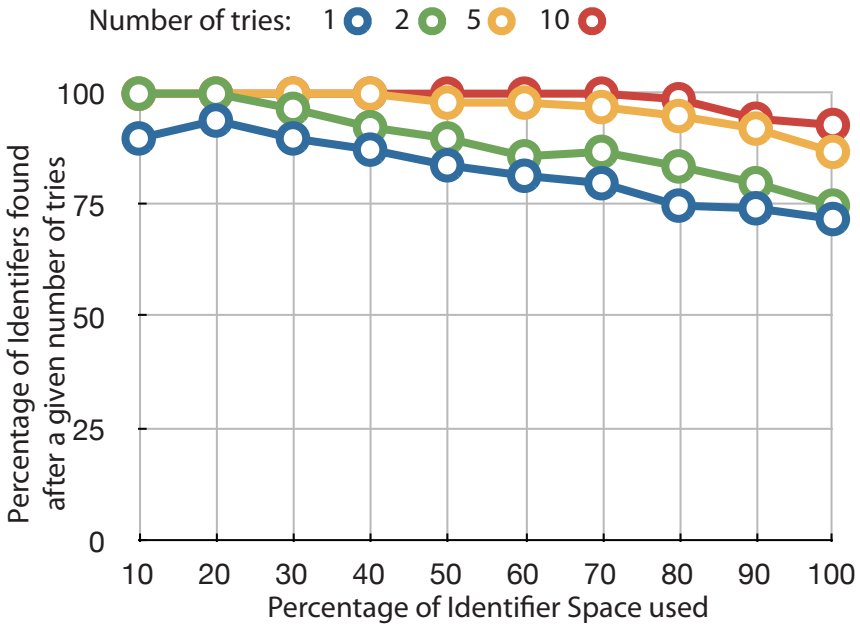

Figure 12: Graph of the second evaluation, percentage of identifiers allocated after a given number of tries.

already a large percentage of unique identifiers can be found with one single try, but as the utilisation of the identifier space increases, less identifiers can be found. After five tries, almost all identifiers can be found, at leas up to an utilisation of around 70%. After ten tries, all identifiers could be found up to an utilisation of 80%.
This experiment shows that ten tries is a reasonable choice. Although the API doesn't provide the ability of changing this value, the user has some influence on it by changing the AIA timeout value. Regardless of the number of tries already done, AIA stops after the specified timeout.

6 Future Work

6.1 Improved Identifier Generation

The current implementation of AIA, or the Perl script in particular, randomly generates a string that matches the given regular expression. However, it might be preferable that an identifier that's generated for a certain instance of an application is partially reproducible. This could be done by using the identifier or some other unique information from the underlying compartment as a seed. For example it would be desirable that the IP address that belongs to a certain Ethernet interface is similar every time AIA is used to generate an IP address, in this case the Ethernet MAC address could be used as a seed. However, it shouldn't generate exactly the same identifier each time, since AIA is based on the idea that if an identifier is not available anymore another one is generated, and so on. This wouldn't work if always the same identifier was generated.

6.2 Duplicate Identifiers

Even though, we argue in Section 4.5 that the probability of duplicate identifiers is quite low if certain properties are fulfilled, it would be desirable to further decrease the danger of duplicate identifiers. Some possible improvements could include:

Detection Mechanism Applications and protocols could be asked to provide a mechanism that can detect duplicate identifiers, and in case of a conflict resend the identifier request to get a new identifier. However it should be prevented that both instances of the application re-request a new identifier and the possibility of duplicate identifiers is still around.

Verification It's possible to provide another API function in `aia.h` called `verifyIdentifier`, which would just check whether an identifier is really unique after it was published. This functionality could be wrapped in a modified `publish` function:

```
anaLabel_t publishIdentifier(char* regex,
        struct service_s* service_of_underlying_compartment,
        struct service_s* service_to_publish,
        struct context_s* context_where_to_publish,
        struct timespec* timeout_for_aia);
```

This function could request an identifier, publish it and afterwards verify its uniqueness.

Improved Identifier Generation As mentioned in Section 6.1, identifiers could be generated influenced by a seed coming from the identifier of the underlying compartment. Therefore the probability that the same identifier is generated on two different nodes is further decreased.

6.3 Publishing AIA

In the current implementation, AIA is only published in underlying compartments on a given node once AIA was requested to find an identifier in these compartments. This makes sense as long as nodes connecting multiple compartments are not very often, and preferably request identifiers first.
It would be possible to regularly publish AIA in all compartments available on a node. As long as AIA is only used for few types of compartments this might be an overhead. However, if AIA was used to find identifiers for many different compartments, this way of publishing AIA might be a better choice.

7 Summary

In this semester thesis, we designed and implemented a generic and distributed identifier allocation system, which is able to provide unique identifiers to any kind of application or protocol. With this Autonomic Identifier Allocation, the ANA project gains on autonomy.
IP networks can now be set up much easier by using the zeroconf option of `ipconfig`. Nodes can join existing networks without the need of manual configuration or a centralised server providing addresses. But AIA cannot only be used to allocate IP addresses, it provides an API that can be used by every brick within ANA. Identifiers can be allocated uniquely in a compartment that is built over different underlying compartments thanks to the forwarding functionality built into AIA.
However, there is the danger of duplicate identifiers. It has been showed, that this danger can be kept low as long as the address space is used rather sparely and as long as identifiers request do not occur too frequently. In fact, while evaluating and verifying AIA, I never encountered the problem of duplicate identifiers even though also multiple requests have been started at the same time.
The contributions of this thesis to the ANA project can be summarised as follows:

- Before starting with the work on an own identifier allocation systems, I studied and compared different existing projects dealing with automatic distributed identifier allocation.
- The main part of the work on this thesis consisted of the design of AIA. Many options and possibilities of an identifier allocation systems have been considered. Whereas the decision was made for a system as generic as possible.
- In order to be able to efficiently work on the thesis I set up an environment of multiple virtual machines building a virtual network.
- Once planned, the project was implemented in C in the ANA framework. The result is a properly working generic application.
- The chat and the ipconfig applications were changed to use AIA's functionality if needed.
- The project was verified and evaluated in different scenarios with AIA used for different applications.

I hope that in the future many bricks will use the functionality provided by AIA and that autonomic identifiers are only the first step of completely autonomously configured systems or even networks.

References

[1] ANA Autonomic Network Architecture. http://www.ana-project.org, (7.1.2009).

[2] Zeroconf working group. http://www.zeroconf.org/, (7.1.2009).

[3] Apple Inc. *Bonjour*. http://developer.apple.com/networking/bonjour, (7.1.2009).

[4] A. White and A. Williams. Unique identifier allocation protocol. Internet-Draft draft-white-zeroconf-uiap-01, Internet Engineering Task Force, October 2002. Expired on May 1, 2003.

[5] Mesut Günes and Jörg Reibel. Ein dynamisches Adressierungsverfahren für mobile Ad-Hoc Netze. In *Mobile Ad-Hoc Netzwerke, 1. deutscher Workshop über Mobile Ad-Hoc Netzwerke WMAN*, volume 11 of *LNI*, pages 59–78. GI, 2002.

[6] S. Cheshire, B. Aboba, and E. Guttman. Dynamic configuration of IPv4 link-local addresses. RFC 3927, Internet Engineering Task Force, May 2005.

[7] D. C. Plummer. An ethernet address resolution protocol. RFC 826, Network Working Group, November 1982.

[8] Stefan Schmid Christophe Jelger. ANA Blueprint, 2008. http://www.ana-project.org/deliverables/2007/ana-d1.456-final.pdf, (7.1.2009).

[9] Perldoc (perldoc.perl.org). perlre - perl regular expressions. http://perldoc.perl.org/perlre.html#Regular-Expressions, (7.1.2009).

[10] Christophe Jelger. The compartment finder. In *Developmentof autonomic applications*, pages 12–16, 2008. http://www.ana-project.org/deliverables/2007/ana-d4.3-final.pdf, (7.1.2009).

[11] Richard Gold, Per Gunningberg, and Christian Tschudin. A virtualized link layer with support for indirection. In *FDNA '04: Proceedings of the ACM SIGCOMM workshop on Future directions in network architecture*, pages 28–34, New York, NY, USA, 2004. ACM.

A How To

A.1 Starting and Using AIA

All AIA code is located in the directory `./C/bricks/aia/`, this also includes some scripts that start AIA with a set of standard bricks, especially the scenarios mentioned in Section 5. However, in this section we will provide a step by step guide to start AIA for certain scenarios. All commands should be executed from the ANA root directory. But before AIA can be loaded, ANA has to be compiled with AIA. Thus the `config.txt` file needs to be changed:

- The line `USER_PROCESS_BRICKS` should contain at least:
  `ip eth-vl`
- The line `USER_PLUGIN_BRICKS` needs to contain at least:
  `vlink ip chat cfinder aia gatesPlug eth-vl`

After having added all needed bricks to the config file, ANA needs to be compiled using `make`.

A.1.1 Chat Over Ethernet

The following commands start a chat application that runs over Ethernet. In order to be able to use the text user interface for the chat, two console windows are needed. On the first one enter:

```
# start MINMEX
sudo ./bin/minmex
```

On the second console, start vlink, Ethernet, AIA and the chat application with the following commands:

```
# load vlink brick
./bin/mxconfig load brick ./so/vlink.so
# create new vlink
./bin/vlconfig create 1
# add interface eth1 to vlink
./bin/vlconfig add_if vlink1 eth1
# switch on vlink1
./bin/vlconfig up vlink1
# load Ethernet brick
./bin/mxconfig load brick ./so/eth-vl.so
# load AIA brick
./bin/mxconfig load brick ./so/idalloc.so
# load chat brick with AIA functionality
./bin/mxconfig load brick ./so/agnostic_chat.so aia
```

`eth1` has to be replaced by the name of the Ethernet interface of your machine. The chat application should be started on multiple machines and can then be used in the first console window.

A.1.2 Zeroconf (AIA for IP Addresses in Ethernet Compartments)

AIA can be used to allocate IP adresses from the Zeroconf range in Ethernet compartments. This is done as follows:

```
# start MINMEX
sudo ./bin/minmex &
# load vlink brick
./bin/mxconfig load brick ./so/vlink.so
# create new vlink
./bin/vlconfig create 1
# add interface eth1 to vlink
./bin/vlconfig add_if vlink1 eth1
# switch on vlink1
./bin/vlconfig up vlink1
# load cfinder brick
./bin/mxconfig load brick ./so/cfinder.so
# load gates plugin
./bin/mxconfig load brick ./so/gatesPlug.so
# start Ethernet as a standalone process
sudo ./bin/eth-vl -n unix://<SOCK_FILE> &
# load AIA brick
./bin/mxconfig load brick ./so/idalloc.so
# load IP bricks
./bin/mxconfig load brick so/ip_enc.so
./bin/mxconfig load brick so/ip_sum.so
./bin/mxconfig load brick so/ip_fwd.so
# start ipconfig as standalone process with zeroconf option
sudo ./bin/ip_cfg -n unix://<SOCK_FILE> -e eth01 -z &
```

Where `<SOCK_FILE>` stands for the newest of the files in the `/tmp/` directory named like `anaControl_gatesPlug_*`.

A.2 API Usage

Section 4.8 describes the API of AIA. In this section we will describe how the API is used at the example of the ipconfig and the chat application.

A.2.1 Ipconfig

In order to automatically allocate IP addresses to Ethernet interfaces as in [6], we adopted the IP configuration brick to use AIA. In this case ipconfig allocates IP addresses from the network 169.254.0.0/16 like Zeroconf. But in contrast to Zeroconf, IP addresses are unique among all ethernet compartments that are connected to the requesting node by a node that previously assigned IP addresses with AIA. See Section 4.4 for a more detailed description of unique identifiers in disjoint compartments.

```
ip_cfg -e <ethernet interface> -z
```

A.2.2 Chat

As an example that AIA cannot only be used for IP addresses, we also provide a modified version of the chat application which can be used to find unique names throughout a set of compartments. If the chat application is started without a user name passed as argument, AIA is automatically used to find a chat name that is unique in the `eth01` compartment.
In addition when calling the chat application, it can also be specified which protocol needs to be used an in which compartment the user name needs to be unique.

```
agnostic_chat [aia [<interface> [<compartment>]]]
```

For example:

```
// find a user name in eth01
agnostic_chat
agnostic_chat aia
agnostic_chat aia eth01

// find a user name in eth01 but
// unique in all ethernet compartments
agnostic_chat aia eth01 wired

// find a user name in IP
agnostic_chat aia ip

// find a user name in the compartment identified by the IP
// 10.0.0.1 but unique in all IP compartments
agnostic_chat aia 10.0.0.1 ip
```

B Definition of Task

ETH
Eidgenössische Technische Hochschule Zürich
Swiss Federal Institute of Technology Zurich

TIK Institut für Technische Informatik und Kommunikationsnetze

Semester Thesis

Zeroconf for ANA network compartments

Mathias Fischer

Advisor: Ariane Keller, ariane.keller@tik.ee.ethz.ch
Co-Advisors: Theus Hossmann, hossmann@tik.ee.ethz.ch
Dr. Martin May, may@tik.ee.ethz.ch
Professor: Prof. Dr. Bernhard Plattner, plattner@tik.ee.ethz.ch

September 2008 - Januar 2008

Introduction

This semester thesis is in the context of the ANA project. The goal of the ANA project is to explore novel ways of organizing and using networks beyond legacy Internet technology. The ultimate goal is to design and develop a novel network architecture that can demonstrate the feasibility and properties of autonomic networking. One area where autonomic features can be applied successfully is addressing. In this semester thesis we develop an autonomic configuration protocol for network compartments.

Assignment

This assignment aims to outline the work to be conducted during this thesis. The assignment may need to be adapted over the course of the project.

Objectives

The objective of this semester thesis is to design and implement a protocol that autonomously configures ANA network compartments and assigns addresses to individual nodes. Since in todays network the IPv4 protocol is in widespread use we will focus on IPv4 but the protocol should be generic enough to work with any other addressing scheme.

Tasks

This section gives a brief overview of the tasks the student is expected to perform towards achieving the objective outlined above. The binding project plan will be derived over the course of the first three weeks depending on the knowledge and skills the student brings into the project.

Familiarization

- Study the available literature on ANA [1, 2].
- Study the available literature on zero configuration protocols [3, 4].
- Study the available literature on IP/RIP on ANA [5].
- In collaboration with the advisor, derive a project plan for your semester thesis. Allow time to study related work and to develop, implement and validate your protocol. At the end of your semester thesis you will need some time to write your documentation and prepare the presentation.

Protocol Design

- List all elements that need to be configured.
- Derive a protocol that works for IPv4 nodes that belong all to the same underlaying compartment. Keep in mind that it should work for other network compartments as well.
- Optional: Generalize your protocol to work with other network compartments.
- Optional: Enhance your protocol to work within the whole network.

Software Design

- The software should be as generic as possible.
- Divide your protocol into network compartment specific and unspecific bricks.
- Think about possible test scenarios for the functional verification.

Implementation and Validation

- Implement the core functionality of your protocol.
- Optional: Implement additional functionality of your protocol.
- Optional: In several iterations optimize your implementation.
- Validate the correct operation of your implementation.
- Provide a simple validation script, that determines whether your Bricks work correctly.

- Check the resilience of the implementation, including its configuration interface, to uneducated users.
- Document your code with doxygene [8] according to the ANA guidelines.
- Adhere to the Linux coding style guide [6].
- Optional: Try whether your implementation works also in the Linux kernel space.

Deliverables

- Provide a "project plan" which identifies the mile stones.
- Mid semester: Intermediate presentation. Give a presentation of 10 minutes to the professor and the advisors. In this presentation, the student presents major aspects of the ongoing work including results, obstacles, and remaining work.
- End of semester: Final presentation of 15 minutes in the CSG group meeting, or, alternatively, via teleconference. The presentation should carefully introduce the setting and fundamental assumptions of the project. The main part should focus on the major results and conclusions from the work.
- End of semester: Final report describing the semester thesis.
- Use the default ANA layout for all your figures.
- Any software that is produced in the context of this thesis and its documentation needs to be delivered before conclusion of the thesis. This includes all source code and documentation. The source files for the final report and all data, scripts and tools developed to generate the figures of the report must be included. Preferred format for delivery is a CD-R.

Organization

- Student and advisor hold a weekly meeting to discuss progress of work and next steps. The student should not hesitate to contact the advisor at any time. The common goal of the advisor and the student is to maximize the outcome of the project.
- The student is encouraged to write all reports in English; German is accepted as well. The final report must contain a summary, the assignment and the time schedule. Its structure should include the following sections: Introduction, Background/Related Work, Design/Methodology, Validation/Evaluation, Conclusion, and Future work. Related work must be referenced appropriately.
- The source code will be published under the ISC license.

References

[1] ANA Core Documentation: All you need to know to use and develop ANA software. Available in the ANA svn repository.
[2] ANA Blueprint: First Version Updated. Available from the ANA wiki
[3] Inside AppleTalk, Gursharan S. Sidhu, Richard F. Andrews, Alan B. Oppenheimer,
developer.apple.com/MacOs/opentransport/docs/dev/Inside_AppleTalk.pdf (04.09.08)
[4] Zero Configuration Networking (Zeroconf), http://www.zeroconf.org (04.09.08)
[5] Stephan Dudler: New Protocols and Applications for the Future Internet, Master Thesis [MA-2007-39], ETH Zurich
[6] Available on your Linux box: file:///usr/src/linux/Documentation/CodingStyle
[7] https://www.ana-project.org/wiki
[8] http://www.stack.nl/∼dimitri/doxygen/

C Project Plan