UNIVERSITY OF OSLO
Department of Informatics

MCIS - A self-optimizing, distributed information sharing system for the future Internet

Master thesis

Hans Vatne Hansen
hansvh@ifi.uio.no

November 16, 2009

Abstract

This work demonstrates how we designed, implemented and tested a distributed information sharing system called MCIS in a novel, autonomic network architecture. We modify a legacy framework for multiple attribute, range based DHTs and extend it with autonomic features to comply with the requirements and principles of the Autonomic Network Architecture project and our goals for the MCIS.

Our main research focus is on self-optimization through what we call resource adaptation. Resource adaptation is a dynamic calibration scheme where adjustments are made internally in the MCIS in order to cope with varying resource consumption in the system it is running on. We have designed the necessary components for both the MCIS and the resource adaptation extension and made them work together in order to provide a fully autonomous, self-optimizing information sharing system.

Measurements in a real system setup are used to evaluate our self-optimization scheme with respect to throughput and response times. Several machines are configured in a distributed environment where two MCIS nodes cooperate to administer one client inserting data and performing queries. Synthetic load is introduced to consume system resources and consequently invoke the resource adaptation mechanisms within MCIS.

The differences between MCIS with and without self-optimization are clearly noticeable in the query responses. MCIS is able to handle a significantly higher number of simultaneous queries in almost all our experiments, and this increased throughput does not affect the response times negatively.

Our results show that the resource adaptation is most effective when the number of stored data elements in MCIS is large or when many queries are made simultaneously. We have been able to improve the query success rate from 39% to 96% when 2000 data elements are stored and 100 concurrent queries are made per minute, and from 19% to approximately 90% when 1000 data elements are stored and 250 concurrent queries are made per minute. It is evident that self-optimization through resource adaptation is a valuable extension to MCIS.

Acknowledgments

First and foremost I would like to express my deepest gratitude to my two advisers, Professor Dr. Vera Goebel and Dr. Matti Siekkinen, for their invaluable guidance. Without your support, expertise and feedback I would never have been able to finish this work.

I would also like to thank the members of the Autonomic Network Architecture project and all the helpful people at the Distributed Multimedia Systems group at the University of Oslo, especially Professor Dr. Thomas Plagemann and Daniel Rodriguez Fernandez. Your assistance has been greatly appreciated.

Finally, I would like to thank Magnus Oen Pedersen for proofreading this thesis and for all the great times we have had working together on projects, mandatory assignments and eating pizza troughout the last seven years.

Thank you all so very, very much.

Hans Vatne Hansen
University of Oslo
October, 2009

Contents

List of Figures

List of Tables

1 Introduction

1.1 Motivation and background

The Internet today evolves in a way where ad-hoc solutions and patches are common. Autonomic networking is a concept where self-managing networks are formed to overcome the growing complexity of networks in general, and the Internet in particular. In an effort to investigate this concept further, universities and research institutes from Europe and Northern America have formed the *Autonomic Network Architecture (ANA)* project which intends to create a new network architecture from scratch. This new architecture can discover problems by itself, and try to fix them without human interaction.

This ability to operate without user input or assistance consists of different self-star properties, where self-configuration is the key property in an architecture like ANA. Self-configuration means that the network is able to automatically negotiate addresses and protocols and form a well-functioning network. Other important self-star properties are self-protection and self-optimization. Traditional network architectures support some of these properties, but require supervision. The ANA project aims to change this.

One of the main objectives in ANA is to react to changing states and dynamically re-organize the network. This includes switching protocols and algorithms at run time without interfering running services. To accomplish this, several alternatives to any operation must be investigated and some form of monitoring is required. In the classic sense, monitoring is almost always designed for a special situation like failure notification. In ANA, monitoring is a more generic, core component of the architecture, and the self-star properties mentioned above depend upon it to work properly.

We understand that monitoring is very important in network management. The term refers to observing and can be defined as "something that serves to remind or give warning"[12]. In traditional networking, this definition seems to fit nicely. Monitors are usually designed for one such specific task, often after the other network services are implemented. This gives the traditional monitors a priori knowledge about the network topology and make them quite robust, but also make them so specialized that reusing them is difficult or even impossible. Every monitor designed this way has to be written, adjusted and tested for each specific monitoring task. Co-operation between these monitors is also rare[8].

But monitoring must exist in any autonomic network architecture. None of the defining self-star properties can be obtained without monitoring, because they are achieved by a feedback control system driven by monitored events. Knowledge about changes is vital when trying to adapt. As a consequence, monitoring is integrated into ANA as a first class citizen, just as addressing and naming. Monitoring information can help routing optimization, service placement and cooperation between functional blocks and typical objectives for monitoring include failure detection, quality of service, assurance, service level agreement compliance, and traffic engineering[8].

Because the monitoring data can differ greatly in content and have many different uses the underlying database must comply to a vast range of requirements. Most importantly, it can not confine the data to only some uses so other uses are excluded. The database must also be fast and reliable, even with a large number of users and much data. Lastly, the database must provide reasonable response times and data throughput. To support all these requirements together is a challenge.

To serve many different kinds of data to a large set of peers is not a trivial task. Scalability is a huge field of research with many pitfalls. The main problem is often that the number of users or amount of data becomes too big for the system to handle and this becomes a bottleneck in respect to response times[16]. If a store and query system can detect changes in system resource consumption and make adjustments to itself before the response times increases, it will increase data throughput and improve the quality of the applications using it. Programs capable of doing these kinds of adjustments are called adaptive software and are an important aspect of modern distributed systems[16].

A resource is a system entity required by tasks for processing data. In a standard computer this means the processor, the systems memory, and its storage capacity. Adaptation must therefore be done to optimize the utilization of one or more of these resources. Because hard drives have become very cheap in the recent years, storage is almost never a limiting factor in modern computers. We reason that the resource adaption should focus on the two remaining entities, processor and memory.

1.2 Goals and problem description

The main goal of this thesis is to develop a distributed information sharing system in an autonomic network. We accomplish this by transforming an existing information sharing program called Mercury into a novel system with autonomic features. The core autonomic feature is self-optimization, where our system reacts to changes in processor load by gracefully starting or shutting down internal structures correspondingly. We call this resource adaptation, and we want to investigate if this self-optimization scheme can improve the quality of service of our information sharing system. The whole system consists of four components:

- The main information sharing system,
- a client application using the information sharing system to store and query data,
- a measuring component knowing the exact resource consumption at all times, and
- a component making adaptation decisions based on the measurements and consequently notifying the main system.

These four components need to interact with each other, and the main system should also be able to interact with other functional blocks in ANA wanting to store or query data. Some of these components, like the measuring component and the decision component, could potentially be merged. However, it is an ANA principle to keep different components small and flexible for easy reuse or change. Because this work is partly an effort to gain insight into development of autonomic networks and concepts for the future Internet we choose to comply with this principle.

We test the self-optimization scheme by performing a small scale experiment with the complete setup described above. We perform evaluation tests of the information sharing system where we measure and investigate all the responses given to the client application when resource adaptation is disabled and compare these results with identical tests where resource adaptation is enabled. If significant improvement of the results is visible we have strong evidence that self-optimization of our information sharing system is possible through resource adaptation.

1.3 Contributions

Our research contributions to the area of distributed systems is mainly the development of a information sharing system and a self-optimization extension for it. We have designed and implemented a generic information sharing system called MCIS with capabilities for storing and querying data. We have also designed and implemented two applications responsible for measuring and analyzing a computers resource consumption. These two applications are part of the self-optimization extension and signal the information sharing system when vital machine resources are running low.

All the components we have developed are evaluated with a small scale experiment. Because we have a complete, functioning system we are able to set up a distributed environment running our applications, administer tests on these applications and investigate the impact of the self-optimization scheme. This evaluation is an attempt to prove that self-optimization can be done through resource adaptation.

The development of the ANA core and some of the other components we use is done by other research institutions, giving us a big challenge throughout this work. Understanding the ANA principles and getting into the autonomic mindset takes time. In addition, we have to learn the ANA core and how to utilize the very complex Mercury system.

1.4 Outline

This thesis is divided into eight chapters. The first five chapters provide background information and introduce design concepts relevant to the conducted research. The last three chapters are related to the realization and evaluation of our work.

Chapter 2 is an introduction to the Autonomic network architecture with information about different abstractions used in ANA, and also about the ANA core. Chapter 3 is an introduction to Mercury, the system we have based our information sharing system on. We see how data is inserted and retrieved and how Mercury can be used as a base for distributed applications. Chapter 4 contains information about MCIS, our information sharing system. In this chapter we show how MCIS is designed and how it can benefit from self-optimization. A more detailed description of the self-optimization scheme is given in Chapter 5. It contains information about the different concerns

in resource adaptation, and how we separate these concerns into different applications.

Chapter 6 presents a detailed description of the implementation we have done and the related challenges we have encountered. The chapter is divided into sub sections for each of the components in our system. Chapters 7 demonstrates how the system works and our experiences with it. Evaluation of the results and the research methods is also prestented here. Chapter 8 concludes the thesis with a summary of our contributions, a critical assessment and suggestions for future work. Extended technical documentation is provided in the Appendix and on the attached DVD.

2 Autonomic Network Architecture

A network architecture is the design of a computer network defined by a set of communication principals within a certain scope. The well known *Open Systems Interconnection Basic Reference Model (OSI Model)* is such an architecture. Like most other traditional network architectures it incorporates rigid standards to enforce compatibility between participating nodes. There are many clever solutions in the OSI model, but also some flaws. One example of a problem is the need for a priori knowledge about the subnet in the network layer.

"To achieve its goals, the network layer must know about the topology of the communication subnet (i.e., the set of all routers) and choose appropriate paths through it."[15] Another rigid standard is the global address space in the *Internet Protocol (IP)*, requiring uniqueness and global coordination. To solve problems like these, new protocols and schemes need to be implemented. These new extensions may not be compatible with existing hardware or software and can cause disruptive behavior.

A solution might be to handle the changes when they arise and focus the architecture development on flexibility and autonomic behavior. The term autonomic refers to the systems ability to perform via self-star attributes, meaning without human intervention. The self-star attributes include

- **self-configuration**: set up and maintain components automatically,
- **self-healing**: discover and correct problems,
- **self-optimization**: monitor and control resources and ensure optimal functioning and
- **self-protection**: identify and protect against arbitrary threats.

These attributes give the network the ability to negotiate protocols and related settings automatically, and make a fully working network. They also imply scalability in respect to both size and functionality meaning that new functions can be added, even without interrupting existing services, and that an ever growing userbase is supported. Another positive side effect is that different variants to perform a specific task can be implemented and tested, making it easy to replace or upgrade parts of a system. All these things

combined allow an autonomic framework to evolve and grow when new technologies and requirements emerge.

This is all in total contrast to existing architectures where patches, firewalls and ad-hoc solutions like *network address translations (NAT)* are common. These types of solutions are making communication between different devices and networks diffcult and are becoming a burden for end users as well as system developers. An example of bad scalability in the Internet today is the IPv4 address shortage.

2.1 Overview of ANA

A new, emerging network architecture is the *Autonomic Network Architecture (ANA)*. ANA is currently being developed with the goal to explore novel ways of organizing and using networks, beyond legacy Internet technology[11]. The ANA project has two complementary goals: one scientific, and one technological. The scientific objective is to identify fundamental autonomic network principles. By incoroprating the different self-star properties the hypothesis is that ANA networks will naturally be richer in functionality and scale in size. The technological objective is to build an autonomic network architecture and demonstrate that it works.

Developers from universities and research institutes in Europe and Northern America are working together to reach these goals and design and implement the entire architecture from scratch, based on autonomic principals. No such architecture exist today, so the ANA project is doing innovative work on this research topic.

The ANA project does not envision one static architecture for the all various types of networking scenarios that exist today, and might exist in the future. The intention is to provide a meta architecture that enables the co-existence and interworking between these different networking styles like sensor networks and mobile ad-hoc networks. The work is an attempt to identify needs for the future Internet.

2.2 ANA abstractions

The ANA abstractions are a set of entities providing a common language for networks to interact. There is a consensus in the research community

that one size does not fit all, and that the existing Internet technologies are becoming obsolete[11]. The major problems with existing architectures are that they often rely on a global address space and lack a change management system, as noted above.

2.2.1 Network compartment

Network compartments are similar to overlays and domains, and can be understood as wrappers for different implementations of networks. They have few restrictions on how they work internally, but strict paradigms for how they interact with other networks. Registration and resolution are key functions here. It is worth noting that interaction between compartments can be overlaid so that one compartment uses the communication services of another compartment, and vice versa[2].

Communication between members inside a compartment is done according to some commonly agreed set of principles, protocols and policies that every entity must obey. These operational rules and administrative policies for the communication context include how to join and leave a compartment and how to reach another member. A member can be anything from a node, a set of servers to a software module.

Each compartment is free to use any naming and addressing scheme it wants, enabling custom solutions for each network. Some compartments might not even need addressing, for example in sensor networks. In addition, it enables the use of future solutions. The drawback is that global routing becomes similar to searching.

The communication entities inside a compartment are typically represented as functional blocks, described below.

2.2.2 Information dispatch point

The fundamental concept introduced by ANA is *information dispatch points (IDPs)*. They are startpoints of communication and the entire architecture is built around them. Traditional network architectures have primarily focused on endpoint addresses, but ANA does the exact opposite. ANA mandates that all communication starts at IDPs. IDPs can be accessed by doing a resolve request as specified in the compartment API.

An analogy to the concept of IDP is a telephone number that reaches you where ever you are. The number is decoupled from the actual phone, so instead of always ringing on your home phone, the telephone number points to your office phone when you are at work, your Skype account when you are using your computer and your cellular phone when you are outside. More formally, IDPs are interfaces to underlying processing entities.

A big advantage of using IDPs instead of endpoints is the ability to redirect traffic at run time and handle changing states, as shown in the telephone analogy. This rebinding is transparent to the users and the interface remains operative throughout its entire lifetime. Figure 1 shows a static entry point with a changing functional block underneath. This kind of remapping can be done to update existing entities or replace them completely.

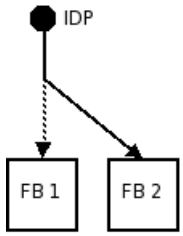

Figure 1: Remapping of functional blocks

IDPs perform no processing except data forwarding, so the actual data management is done in functional blocks. This way IDPs provide a generic communication paradigm and a flexible way to reorganize structure without disruption.

2.2.3 Information channel

An IDP gives access to an *information channel (IC)* which is the link between members in the network. An IC is an abstraction of a communication service provided by an underlying system and can conceptually be seen as the communication medium between functional blocks. In reality ICs are really functional blocks belonging to the network compartment. They can be unicast (sending to a single destination), multicast (sending to several destinations simultaneously) or any other form. The flexibility allows ANA to evolve if and when new transmission technologies arise.

ICs can be either physical in form of a cable, or logical in form of a chain of packet processing elements.

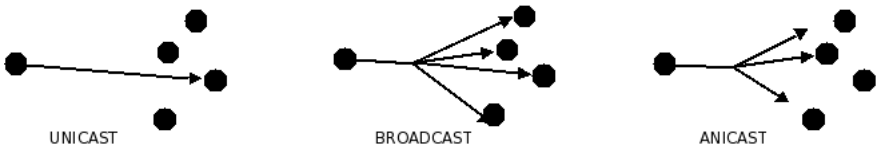

Figure 2: Routing schemes

2.2.4 Functional block

A *functional block (FB)* is a representation of a procedure or data handling algorithm. FBs can generate, consume, process or forward information based on predetermined rules, and it can have several different input and output IDPs. A network monitoring module and a distributed database are examples of things that can be abstraced as FBs. In constrast to the OSI entities, FBs can provide functionality ranging from a full monolithic network stack down to a very small entity computing checksums[2].

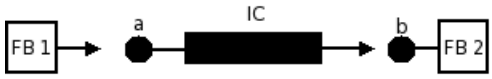

Figure 3: Connecting FBs and ICs through IDPs

Using a FB involves resolving an IDP and sending data to it through an IC. In Figure 3 we see FB 1 sending information via an IC to the FB bound to IDP b.

2.3 ANA core

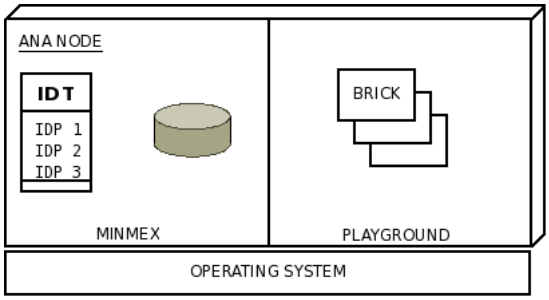

Figure 4: The components of an ANA node

The ANA core is the software implementation of ANA. It has two main components; The MinMex and the Playground, on the left and right side in Figure 4 respectively. The Playground is the execution environment where all the bricks in a node are located and the *Minimal Infrastructure for Maximal Extensibility (MinMex)* is the heart of a node.

With heart we mean that it is an essential, but small kernel-like process supporting the ANA API. It is used by bricks to find and communicate with each other, in addition to utility functions such as garbage collection. Bricks always use the MinMex to pass messages and never communicate directly. This is crucial since IDPs can change their underlying FBs any time.

The individual components of the ANA playground are called bricks. They are inspired by *Component-based software engineering (CBSE)* which is a software engineering discipline with emphasis on decomposition. Software components have well-defined interfaces and produce a specific event or service.

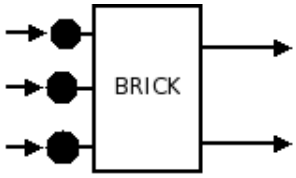

Figure 5: Brick

In Figure 5 we see a brick with three input interfaces and two output interfaces. What the brick does internally is not important or necessarily known, but what parameters it needs to function and what type of feedback it produces afterwards is always well-defined. The key properties of bricks are

- different uses,
- not context-dependant,
- usable with other bricks,
- encapsulated and
- independent of versioning.

A brick is an implementation of a FB, and can be alone, or as a member of a bigger FB implementing a compartment. We understand that bricks have different uses and that they are often used together with other bricks.

One way to visualize the ANA playground and the bricks within is as a set of Lego constructions. Large Lego constructions are assembled using many different bricks, and one specific brick can be used in many different constructions. Some bricks might even be replaced by others as the developer or builder sees fit. Afterwards, the pieces can be taken apart and used to form new constructions.

The bricks use various gates such as UDP sockets, named pipes, generic netlink and shared memory for communication with the MinMex.

3 Mercury

Several protocols for providing scalable peer-to-peer systems have been proposed in the recent years. Most of them, like Chord and Pastry, are based on *distributed hash tables (DHTs)* and suffer from the limitations that these systems incorporate[16]. Their hashing algorithms and underlying key-space solves the problem of load balancing in an elegant way, but prohibits them from offering multiple attribute and range based queries. A multiple attribute, range based system allows a user to specify several keywords within a range of desired values. In a file sharing application this can be $size < 100$ *MB*, $type = MPEG$ and $name = *$. This query returns all MPEG files which are smaller than 100 megabytes. It is a very flexible way of searching and is useful both for answering user queries and in the construction of distributed applications. DHTs implement a strict (key, value) look-up service where these types of queries are not possible. However, there are some key features in a DHT system that are useful in almost all distributed systems[14].

- Decentralization: Participating nodes are both clients and servers forming the system together, without any central coordination.
- Scalability: The total system resources are the sum of all nodes' resources, and it performs equally good or better when more nodes join.
- Fault tolerance: The system does not loose data even when nodes suddenly fail or decide to leave.

Mercury is a distributed system similar to a DHT. It has the three mentioned capabilities in addition to load balancing and support for multiple attribute, range based queries. In addition, it is able to provide logarithmic-hop routing and near-uniform load balancing[5].

Another interesting feature of Mercury is its structure. Mercury is designed as generic system template with a very modular structure, allowing developers to use it as the core of almost any distributed application. It is also possible to replace the underlying network architecture to both a simulation environment, emulating an entire network on one machine, and using ANA as the network architecture. Implementing ANA bricks utilizing capabilities of Mercury is therefore possible. We will take a closer look at how this is done, but first we need to investigate how Mercury works.

3.1 Design of Mercury

The Mercury developers argue that multiple attribute, range queries can enhance search flexibility in a number of scenarios. They point out that DHTs offer a number of scalability advantages, but that the hash table is not flexible enough for many applications. Mercury is designed similarly as DHTs, but have some fundamental changes.

A Mercury application is a system with collections of nodes called attribute hubs, much similar to a DHT overlay. A separate, logical hub is created for each attribute in the application, and nodes participate in several of these hubs. Inside the hubs, nodes are organized in a circular manner giving each node responsibility for a range of values corresponding to the attribute of the given hub. The data is stored contiguously. Using the same file sharing example as earlier there could potentially be three hubs, H_{size}, H_{type} and H_{name}.

Each node maintains three different sets of links.

- A few next and previous links to other nodes in the hub (to prevent system failure when a node leaves the system),
- a few of long-distance links, known as fingers in the literature, to other nodes in the hub (for faster routing),
- and one link to each of the other hubs in the system (cross-hub links).

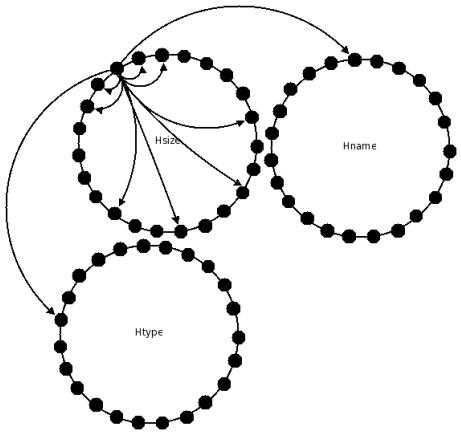

Figure 6: Typical set of links for one given node

3.1.1 Storing and placement of data items

A key feature of Mercury applications is storing of data items. Different types of applications can have different types of data items they want to store. For example, a first person gaming application needs to store information about player health and coordinates, and a file sharing application needs to store metadata about the shared files.

In a Mercury application, data elements are stored in typed attribute value pairs like *int size = 37*. This means that the attribute name is size, its type is integer, and its corresponding value is 37. A data element can have several of these value pairs, much like a record in a relational database. In Figure 7 we see a visual representation of the data element fields corresponding to the three hubs from the file sharing example.

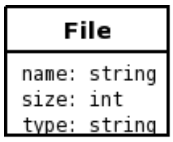

Figure 7: Data element in Mercury

If the data element has multiple attributes, like in Figure 7, the actual element will be stored in a couple of nodes while the remaining nodes hold pointers to it. This gives the Mercury application the opportunity to pass queries to any of the hubs with matching attributes and getting a valid result back.

Unfortunately, because of the way Mercury is implemented, this placement algorithm clusters data around certain nodes and hubs more than others. In a DHT, the randomizing hash function distributes the data evenly, but Mercury needs to do explicit load balancing to cope with this partitioning. We will take a close look at how Mercury solves this later.

The actual storage facility in Mercury can be persistent using an SQL database or volatile in RAM. We have chosen to use the latter to make sure I/O is not a bottleneck for our system. The side effects of this choice are increased RAM consumption, and that data loss is possible if a power failure occurs.

3.1.2 Retrieval of data items

The other key feature of a Mercury application is retrieval of the stored data items. This is done through queries. A query is constructed by a conjunction of ranges in one or more attributes, and passed to either of the hubs with matching attribute responsibility. If an application sends the query "*size* $<$ *100* and *type* $=$ *MKV*" to Mercury, it will be sent to either the hub responsible for size or the hub responsible for type, and then routed to the node with the appropriate value range. Unspecified attributes in queries are considered to be wildcards and match all entries. In this case the query will likely be forwarded to the H_{type} hub because it is estimated to be most selective, i.e have the smallest value range. This way, most of the routing is done within an attribute hub and a query is never flooded to an unnecessary large number of nodes.

In Figure 8 we see how a data item is inserted into all the different attribute hubs and how a potential query could retrieve the data from one of the hubs afterwards.

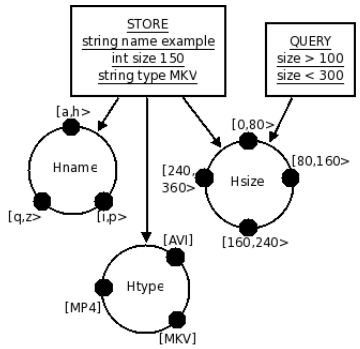

Figure 8: Storing and querying in Mercury

The replication of data items, combined with the structure of the hubs provide low query response times[5], but even if response times are good, there might still be room for improvement.

3.1.3 Load balancing

Maintaining state information about a large set of nodes in a distributed network is not easy. Mercury uses sampling to gather as much information about its system as possible. A node wanting a sample sends a request

message with a small *time to live (TTL)* field to a random neighbor. This neighbor decreases the TTL and forwards it to another random neighbor. This continues until the TTL expires and the appropriate node sends back a message with all its state information. These sample messages can be piggy-backed on normal traffic in the system and is used to form load histograms in the receiving node. In turn, Mercury uses the histograms to perform balancing of data elements.

In many applications attributes have a tendency to group around certain values, as described above. This causes a problem in Mercury since it gives the nodes in charge of these value-ranges much more load than other, less populated ranges. To cope with this Mercury uses a load balancing scheme using the sampling histograms to determine existing load in the system. Heavily loaded nodes probe lightly loaded parts of the network to try to find nodes with small value ranges. When such a node is found the heavily loaded node asks the lightly loaded node to join its attribute hub and take charge of half of its values. The lightly loaded node is now a predecessor of the heavily loaded node. This message passing and all the calculations increase CPU consumption.

3.2 Application design using Mercury

Designing distributed applications is difficult. Challenges include dividing responsibility for data elements among the participating nodes in a balanced fashion and meeting performance demands with a high number of simultaneous users. We have seen that Mercury provides scalable, multiple attribute, range-based queries while still being fast and reliable. Using Mercury as a template for application development is therefore one possible way to achieve these features.

Mercury provides an API for creating distributed applications that can store and query data, and we take a closer look at the available classes and functions in Section 6. The API is not well documented, but there are example applications and source code available. An existing application built with Mercury is Colyseus, a distributed architecture for interactive multiplayer games[4].

In Figure 9 we see how an example application utilizes Mercury in order to store and query data. The application relies on Mercury to perform all the data placement and routing of queries and the two boxes in the Mercury sec-

tion represent classes for doing this. When executing the example application the underlying Mercury software becomes transparent to the user.

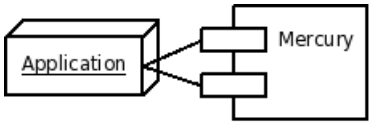

Figure 9: An application utilizing Mercury classes

In order to use Mercury as a base for a new information sharing system in ANA there are some alterations that need to be made to the Mercury software. Features like scalability and fault tolerance are already in place, but other ANA abilities, like self-optimization must be implemented by us. Mercury based applications are also envisioned to be used with the old Internet architecture, and as a consequence IP and port numbers are used as identifiers. We need to modify parts of Mercury and replace these identifiers to be able to use it in ANA. Because of the modular structure of Mercury it is possible to separate these parts and change them to ANA labels. This is a good reason to choose Mercury as the base of our system. Another extension that needs to be made is the support for strings. Mercury only supports integers, and an information sharing system like the one we are developing must support at least strings, and possibly other data types as well. This also requires modifications to several parts of Mercury.

4 Multi-Compartment Information Sharing

Certain applications would greatly benefit from range query capabilities and the research community has developed extended DHTs to provide this desired feature. These DHTs can use order-preserving hash functions, or perform no hashing at all, like in Mercury.

The Multi-Compartment Information Sharing (MCIS) system is one such application. It is built as an application on top of Mercury as shown in Section 3.2 and can therefore support multiple attribute, range queries. In addition to these features, load balancing is performed to ensure evenly distributed load. In MCIS, as in Mercury, annular hubs are formed for each of the attributes in the application schema.

The name MCIS implies use by different compartments. This is a key feature, because the need to store and look up data is in broad demand, especially in ANA. Uses for information sharing emerge everywhere, e.g. in monitoring, where FBs need to coordinate data, and in content distribution. The idea is that any FB can provide data for another FB even if they work on different tasks, but both gather the same type of information for different purposes.

In Figure 10 we see examples of different bricks, including monitoring and content distribution bricks, using MCIS for storing and querying. MCIS does not care what the different bricks do, what other bricks they are connected to or what kind data they would like to store. MCIS offers a information sharing service that bricks can use if they need a storage facility or to distribute data. For example, the monitoring brick can store data that the video on demand brick can use.

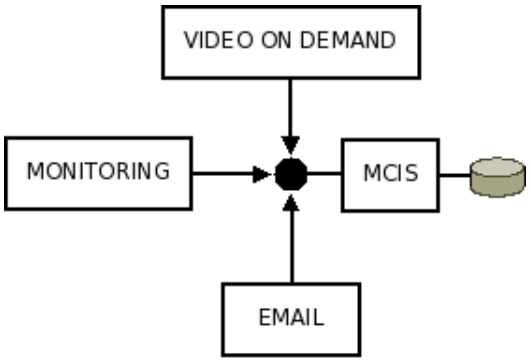

Figure 10: Different bricks using MCIS

4.1 Compartments

The MCIS is composed of one common metadata compartment and one data compartment for each of the data types stored in the system. Each participating node becomes a peer in the metadata compartment together with the other nodes. This metadata compartment provides access to the underlying data compartments and is the entry point for third party storing and querying. There is no imposed hierarchy between the compartments except that all nodes that are part of any data compartment are naturally part of the metadata compartment.

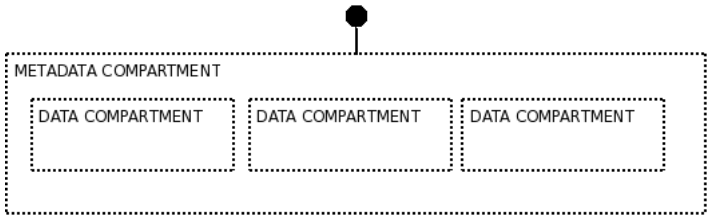

Figure 11: Compartments in MCIS

Every compartment, including the metadata compartment and all the data compartments, run one Mercury instance each.

4.1.1 Metadata compartment

A common metadata compartment for the entire system enables the discovery of the different data compartments. The metadata compartment is managed by a set of MCIS nodes, each responsible for a range of data compartments. The metadata compartment consists of a Mercury attribute hub with data compartment identifiers.

There are three compartment API primitives available in the metadata compartment. However, the unpublish function is not finished in the current version of MCIS, but it might be implemented in the future.

- **Publish**: Start a new data compartment.
- **Unpublish**: Shut down a data compartment.
- **Resolve**: Locate a data compartment and retrieve the IDP to it.

4.1.2 Data compartment

All nodes that store data are part of at least one data compartment in addition to the metadata compartment. A data type in MCIS is represented as several attribute value pairs like *int size = 37, string type = AVI* and the specification of the collection of these attributes is called a schema. Each of the data types have their own data compartment which function and organize their data independently from each other. MCIS uses one Mercury instance for each data type and, in turn, Mercury uses one attribute hub for each attribute inside. The data elements are replicated and inserted into all of the ring structures while a query is only forwarded to the hub where it is expected to be most efficient. It is not mandatory for hubs to exist for all the attributes in the schema, but attribute hubs help to decrease hop count when many MCIS nodes collaborate. MCIS can start fewer hubs or shut down hubs after a data compartment is started.

MCIS can handle several isolated data compartments and other FBs can reach any data compartment with some of the primitives of the ANA compartment API.

- **Publish**: Inserts a data element into the data compartment.
- **Unpublish**: Removes a data element from the data compartment.
- **Resolve**: Return IDPs to nodes which have a certain data element.
- **Lookup**: Perform a query.

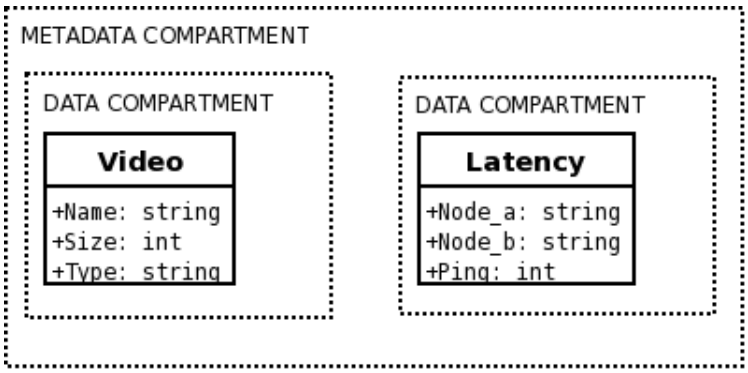

Figure 12: Data compartments in MCIS

In Figure 12 we can see a hypothetical metadata compartment containing two data compartments. One called *Video* with information about different movies, and one called *Latency* containing ping times between nodes. The attributes inside the two data compartments are stored in Mercury hubs.

4.2 Self-optimization in MCIS

Self-optimization is one of the self-star attributes in autonomic systems and a core part of ANA. As we have seen, an ANA environment is a set of autonomous compartments interconnected to form a complex topology[6]. Optimization in such an environment is a difficult task. To make it more manageable, and to comply with a compartment's full autonomy, each compartment will have its own self-optimization mechanisms. This could be internal routing optimization or any other scheme resulting in performance gain.

The drawback of optimizing internally in a compartment is that some mechanisms might influence other compartments involuntarily. In the best case the optimization is inter-compartment aware. This means that the compartment understands the consequences of its optimizations and finds the best solution to improve performance with respect to all compartments.

We know that information sharing between a large set of nodes is difficult and that query responses might fail with increasing load. Research has shown that the main problem of distributed systems is that the number of simultaneous users becomes too high for the system to handle and that this becomes a bottleneck in respect to correct responses [16]. One way to try to solve this problem is with self-optimization. There are many resources in a system like MCIS that could be the source of scalability issues, but we have seen that Mercury handles many of these very good without modifications.

It is still true that if a store and query system like MCIS could detect changes in its environment and make adjustments to itself before the query responses suffer, it could increase data troughput and improve the quality of the applications using it. Adaptive software like this is an important aspect of both modern distributed systems[16] and autonomic networks[2].

When running simultaneously as other processes or on embedded devices, the performance of MCIS might suffer because of it. In Section 5 we will take a closer look at how MCIS can detect changes in the system it is running on

and adapt to these changes.

4.3 Using MCIS in the ANA Integrated Monitoring Framework

Monitoring is a fundamental part of ANA. The self-star properties depend on monitoring and it can help routing optimization, service placement and cooperation between functional blocks. The following sections demonstrate how MCIS can be used to distribute data in the ANA Integrated Monitoring Framework and is one example of the demand for MCIS in ANA.

4.3.1 Overview of IMF

The ANA *Integrated monitoring framework (IMF)* consists of three different types of FBs, shown in Figure 13. They are measuring FBs, an orchestration FB and client FBs. They collaborate to provide interfaces for storing and querying monitoring data, and the main function of the framework is to manage interaction between FBs that produce monitoring data and FBs that use monitoring data, called producer and consumer FBs respectively.

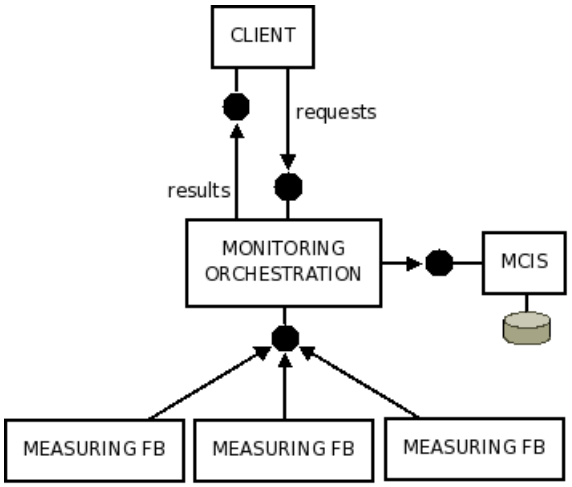

Figure 13: Conceptual view of the monitoring framework

The measuring FBs are producers performing measurements on different metrics, usually system parameters, and send the observed data to the orchestration FB.

The client FBs are consumers wanting information about one or more system parameters. They send requests to the orchestration FB using the following interfaces, and receives data back from it.

- **On request**: The client specifies which parameters to measure when it wants them.
- **On timer**: The client specifies a time interval and will receive periodic measurements based on this value.
- **On condition**: The client specifies a condition (e.g., CPU load $> 90\%$) and receives a notification when the condition is fulfilled.

The orchestration FB is the most important component of the framework and is responsible for all the coordination between participants in it. It has five main functions which are performed by several underlying bricks, all transparent to the client. In addition, it has some control over measurement FBs and can ask them to stop, restart, pause and resume their measurements.

- Keeping track of all the measuring FBs and what kind of information they can obtain.
- Parsing requests from clients.
- Transforming requests into tasks for the measuring FBs.
- Store data into MCIS.
- Providing results back to clients based on observed data.

We understand that the IMF allows different client FBs to apply self-star properties based on observed information. Redundant and inconsistent information is avoided by having one central source of orchestration rather than several independent ones, and new measurement FBs can be added on the fly.

4.3.2 Data management in IMF using MCIS

We have seen that the data gathered by the measuring FBs can differ greatly in content and have many different uses. This means that the underlying data store of IMF must comply to a vast range of requirements. Most importantly, it can not confine the data to only some uses so other uses are excluded. Since the IMF does not impose constraints on what the measuring FBs gather, all kinds of data must be accepted. In addition, the data store must be fast and reliable, even with a large number of queries and huge data amounts. Lastly, it must provide reasonable response times and data throughput.

All the monitoring data needs to be stored somewhere reliable. The storage facility should be stable, fault tolerant and scalable, but still flexible enough not to put too many constraints on the stored data.

The MCIS system is able to provide this link between the measuring FBs and the clients. Because it supports multiple attribute, range queries it makes information sharing easy and versatile. These queries are especially useful in monitoring where various kinds of data is generated by several monitoring processes[8]. The MCIS also improves overall performance of the IMF by self-optimizing. This is done through the resource adaptation scheme shown above. As we have seen, the data in MCIS is stored in RAM, but is replicated to guarantee persistence.

Figure 14 shows an example of a possible data compartment in MCIS for latency measurement. The queries to this data compartment could be *ping* $< 10\ ms$ resulting in a list of node-pairs with low latency, or $node\_a = dmms$ resulting in that particular nodes latency with other nodes.

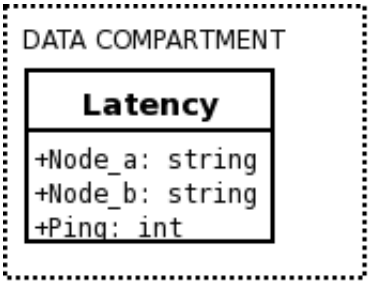

Figure 14: MCIS data compartment for a latency FB

5 Resource adaptation in MCIS

Autonomic behavior is achieved through self-star properties and rely on distributed information sharing. One example is the IMF, which depends on MCIS in order to distribute measurement data across different compartments. Other ANA components, like the VoD FB, use MCIS to store and query data for themselves. All the different uses of MCIS and the corresponding requirements, combined with the overall goals of ANA, lead to self-optimization for MCIS.

Self-optimization means that MCIS is able to react to certain changes automatically and provide a better service because of it. One configuration of MCIS might provide the best service in one situation, where another configuration might perform better under other circumstances. The goal of self-optimization is to identify different situations and make MCIS change itself to work as good as possible under all of them. The environment surrounding MCIS can change continuously and MCIS must function optimally regardless of these changes. Changes include packet loss, but more importantly, the consumption of critical system resources such as processing power[16]. Available processing power is particularly important because MCIS will not be able to perform queries without it.

One possible strategy for self-optimization is identifying when resource consumption is at a level where MCIS is unable to function properly or service its users, adapting to this, and consequently improving performance. We propose to self-optimize MCIS through what we call resource adaptation.

5.1 Design of the resource adaptation scheme

A resource adaptation scheme like the one we propose has different parts and responsibilities. First and foremost, the resource consumption must be measured. If the system does not know how much available resources it has, it is impossible to know when the consumption is at a critical level. A kind of measurement component with detailed knowledge of resource consumption is needed.

Second, the measured resource consumption must be analyzed and correct actions must be taken. Analyzing the available processing power means looking at the measured consumption over time and make assumptions about what

the consequences of these levels are. When a dangerous level of resource consumption is reached, internal configurations must be altered in MCIS in order to cope with the changes. In a similar fashion, when the level of consumption goes back to normal, MCIS is allowed to revert to its previous state. We need an analyzing component with automatic triggers that influence MCIS. The triggers are based on levels of consumption of system resources.

Separating these concerns is coherent with the ANA principal of small, modular bricks. Figure 15 shows a feedback control system for resource adaptation of MCIS with two distinct parts in addition to MCIS itself and a client storing and querying data. Together they dictate how MCIS behaves internally based on resource consumption and predefined trigger thresholds.

The optimal trigger for resource adaptation would be a failing query, but we have to rely on local, a posteriori knowledge. This is because MCIS can not know whether a client application gets its expected query results or not. We argue that processor load should be the trigger for the internal changes in MCIS because there is a direct link between calculating routes in queries and CPU consumption. If there is not enough available processor time, the query might not be routed and will fail.

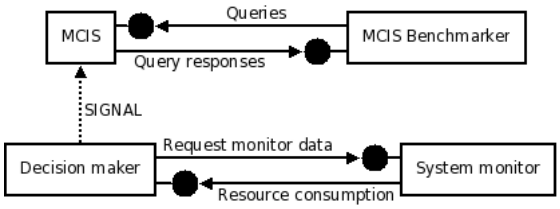

Figure 15: Feedback control system

The different parts of the feedback system and the responsible bricks can be summed up as follows:

- **System monitor**: Measure CPU load and make it available.
- **Decision maker**: Analyze measurement data from the System monitor and make adjustments to influence MCIS based on predefined triggers.
- **MCIS Benchmaker**: Act as a client using MCIS and make log files with detailed information about queries and results.

Analyzing the measured data and triggering the correct actions is the core of self-managing systems[16]. The adjustment made in each MCIS node is whether an attribute hub should be shut down or started. As explained in section 3.1, attribute hubs are where the data is stored in Mercury based applications. By minimizing calculations of where to retrieve data we expect that MCIS is able to answer more queries without failing. This kind of optimization is especially valuable for resource constrained devices like PDAs or systems where resource demanding applications use a large percentage of the available CPU.

An initial configuration is also required. This configuration is not optimized, but reflects a standard system. What a standard system is depends on the device, platform and other running applications. Lastly, some reference triggers are needed. In our system the triggers are predefined thresholds for CPU load where both upper and lower bounds need to be defined.

5.2 Initial configuration

The initial configuration consists of two input parameters. They are controllable to a certain extent, but because they are supposed to reflect an arbitrary system setup, they do not need to be carefully chosen. The parameters are:

- How many attribute hubs each node should start with.
- How much load the processor has before queries are performed.

The goal of these initial parameters is to produce an unoptimized, but working system. The implication of introducing more load should be that the resource adaptation scheme will start optimization, and we can measure its effects.

When it comes to initial hub count there are several choices. We do not know in advance how many hubs a certain setup can handle or how much resources each hub requires. The possible number of hubs will vary depending on different hardware specifications and the external load. One possible initial hub count is the number of attributes in the schema. This is an optimistic approach. It is also possible to be pessimistic, and start with only one hub. A third possibility is to calculate the average between the maximum hubs in the schema and one. Because of the way Mercury is implemented, it is

easiest to choose the first approach. This will probably be close to the correct amount on a standard desktop computer.

The initial processor load will be equal to how much the OS is using together with the minimum of running applications. In a normal situation a resource demanding application can start at any time, but we will explicitly introduce more CPU load anticipating the resource adaptation to start working.

5.3 Decision making and signaling

The only purpose of the Decision maker is to send signals to MCIS telling it to start a new attribute hub or shut one down. A signal is a form of inter-process communication used by operating systems such as Linux, and MCIS has a callback function telling it to start or shut down a hub based on the incoming signal. When the callback function is invoked, the MCIS complies by making the corresponding adjustment.

The signals are not sent arbitrarily, but based on internal criteria in the Decision maker. The combination of resource consumption and the defined thresholds form the basis for the signals. The Decision maker gets its measurements about resource consumption from the System monitor.

We choose which hub to start and shut down based on what we call a *utility score*. The utility score ranks the different hubs based on how profitable they are when performing queries, and is updated every time a hub forwards a message. Each hub score is calculated in the following way: If a hub forwards a data element, the hub utility score is decreased by one. If it forwards a query, the score is reset to zero. We always choose to part from the hub with the lowest score. The reasoning behind this strategy is that forwarded data items add to resource usage, but do not contribute when performing queries. Data items are always replicated and sent to each hub, but queries are only forwarded to the most selective hub, which is then responsible for providing results. I.e. if there are no queries sent to a particular hub, that hub has no value.

In the future the Decision maker can be extended beyond what it does today. It is possible to make more informed decisions, for example by collecting and aggregating data over time. This way trends become more visible and the decisions more resistant to random deviations. Another possible extension is to collect data from different sources. This is redundant when it comes to

CPU load, but can provide additional insights when decisions are based on other criteria.

5.4 System monitor

To be able to react to changes in resource consumption, the data must be collected, stored and made available to inquiring entities. We have seen that there are many possible resources to inspect, but that the main area of interest is processor load.

The System monitor inspects system resources every 5 seconds and stores the observed state both in memory and to log files. This information is obtained from the /proc file system which is a pseudo file system in the Linux kernel used to store system configuration[1]. The disadvantage of using /proc is that it mandates the use of Linux as the underlying operating system. This poses no new limitations in our work because we are already bound to Linux by the Mercury system.

Like the Decision maker, other bricks can query the System monitor at any time asking for the current resource consumption. In addition, the measurements can be retrieved from the log files for later inspection.

A possible extension for the System monitor is to collect more data than just CPU load and RAM utilization. Different decision bricks have different monitoring needs than our self-optimization scheme and the System monitor could potentially know about every system parameter, from memory statistics, logged-on users to which applications are running.

6 Implementation

In the following sections we will take a closer look at how Mercury, ANA and MCIS are implemented and how they work together to form the self-optimizing, distributed information sharing system.

6.1 Implementation overview

A large group of developers are contributing to the bricks used in our research. Some of these bricks, like the System monitor and Decision maker, is designed especially for this thesis, while others, like the IP bricks, are developed by other ANA partners.

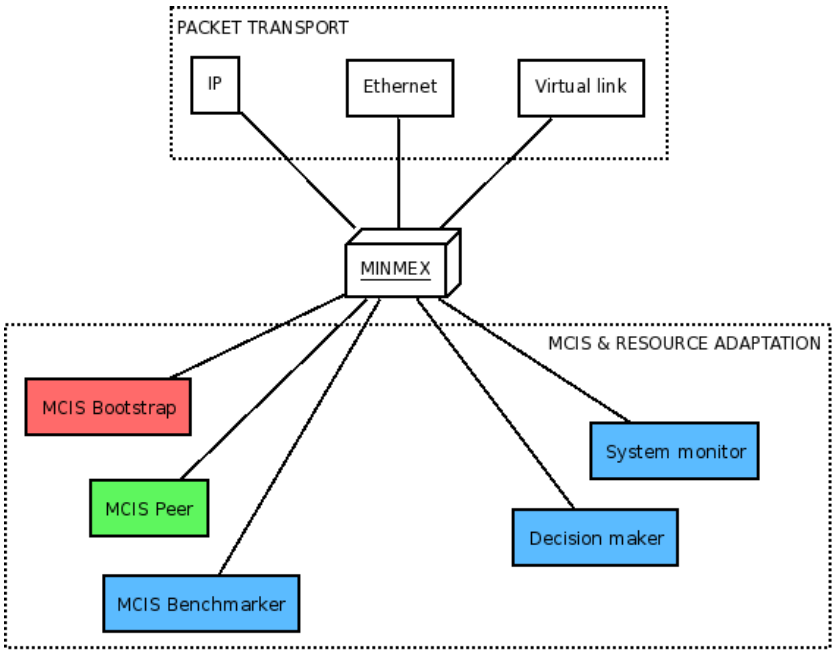

Figure 16: Brick developers

White	Designed and implemented by other ANA partners.
Red	Designed before this thesis. Implemented before this thesis.
Green	Designed before this thesis. Implemented in this thesis.
Blue	Designed in this thesis. Implemented in this thesis.

Table 1: Legend for Figure 16

Figure 16 is an overview of the bricks used in this thesis. The bricks with white fill in this figure are developed by other ANA partners, while the remaining are developed at the University of Oslo. The bricks developed as part of this thesis, are marked correspondingly in the legend. This includes bricks with green and blue fill.

6.2 MCIS Data structures

In Figure 17 we can see the class hierarchy and call structure of Mercury and how the MCIS Peer uses the `Event`, `MercuryNode`, `Constraint` and `PubsubStore` classes to inherit functionality. In turn, these classes depend on other, underlying classes like `Router` and `MercuryID`.

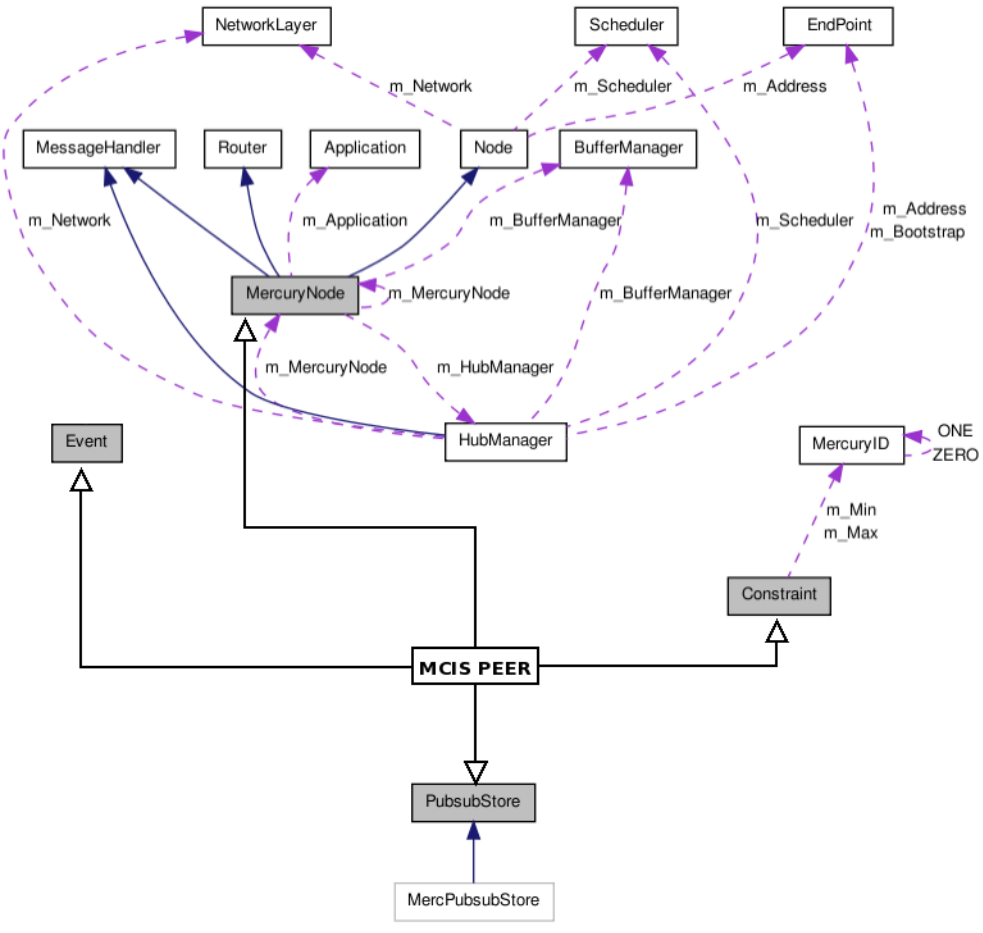

Figure 17: MCIS Data structures

The MCIS Benchmarker, System monitor and Decision maker bricks are not shown in Figure 17 because they do not rely on Mercury. Their implementation is described in detail below.

6.3 Mercury Implementation

Mercury was developed between 2002 and 2005 at Carnegie Mellon University. It incorporates some freely available software components, but the developers have written the entire Mercury system and a distributed game utilizing the Mercury functionality. The source code for the game gives an introduction on how to write Mercury based applications.

The implementation structure of Mercury is modular with eight different directories with distinct tasks and meanings.

- **configs** holds attribute schemas and files containing parameters overriding default values in Mercury,
- **util** contains miscellaneous helper functions like timers, a reference counter and a stack tracer,
- **mercury** is the system core with classes for HubManager, LoadBalancer, BufferManager, Peer etc,
- **wan-env** is the wide area network environment with responsibility for transport.,
- **sim-env** is a simulated environment for doing tests with many nodes running on a single computer,
- **ana-env** is the ANA environment, replacing the WAN environment with the Autonomic network architecture,
- **apps** holds applications utilizing Mercury and
- **build** stores application binaries after compilation.

The environment directories can be interexchanged for different configurations, but we use the **ana-env** for our MCIS application. The **apps** and

build directories will be empty in our project because we will be using Mercury as a part of ANA. Our bricks will be located elsewhere and use the #include directive to gain access to Mercury classes and functionality.

Application development with Mercury is done trough the use of publicly available classes. Most of these classes are not exposed to third party developers, but some are mandatory for all instances of Mercury applications. We will take a closer look at some of the most important Mercury classes utilized in MCIS. Please note that an `EndPoint` is an old Mercury entity and that the name is not accurate. The `EndPoint` class has been rewritten to utilize IDPs for use with ANA.

The `MercuryNode` class is the core of a Mercury instance and is the main coordinator of the underlying system classes like `BufferManager`, `MessageHandler` and `HubManager`. Pseudocode for starting a new data compartment in MCIS looks like the following:

```
ANAMercuryNode *router = new ANAMercuryNode(network, scheduler, address);
router->SetID(data_compartment_name);

for(each attribute in incoming schema) {
  number_of_hubs++;
  join_hubs += router->GetHubManager()->GetAddress();
}

/* Start a MercPubSubStore for saving data in memory */
DummyApp *store = new DummyApp(number_of_hubs);
router->RegisterApplication(store);

/* Start the router thread */
router->FireUp();

/* Initiate handshake with all the hubs in order to join them */
while(router->SendPing(join_hubs) == -1) { OS::SleepMillis(100); }

/* Join all attribute hubs */
while(!router->AllJoined()) { OS::SleepMillis(100); }
```

Figure 18: Mercury class: MercuryNode

The `Constraint` class is responsible for attribute range constraints and have three private variables correspondingly, `m_AttrIndex`, `m_Min` and `m_Max`. A `Value`, or equivalently a `MercuryID`, is the common denominator for attribute values, and can be either an arbitrary string or an integer. The functions in the `Constraint` class are mostly related to retrieving the different constraints.

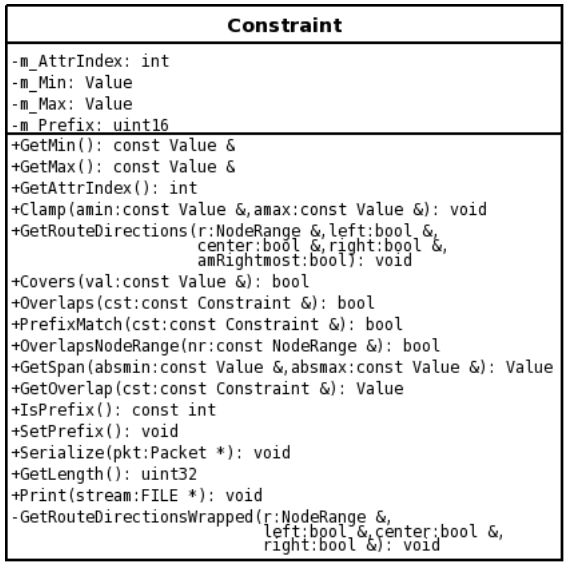

Figure 19: Mercury class: Constraint

The `Query` class is responsible for making queries.

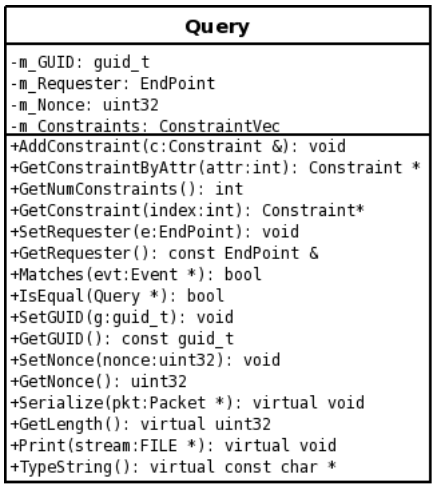

Figure 20: Mercury class: Query

When a query is performed, the `AddConstraint()` function in the `Constraint` class is used to add attribute range boundaries to the query, and the `RegisterQuery()` in the `MercuryNode` is used to deploy the query into the Mercury system. Pseudocode for performing a query in MCIS looks like the following:

```
MercuryNode *router;
Query *query = new Query();

for(each attribute in incoming query) {
  MercuryID min = attribute min;
  MercuryID max = attribute max;
  Constraint boundaries = Constraint(attribute, min, max);
  query->AddConstraint(boundaries);
  delete boundaries;
}

router->RegisterQuery(query);

delete query;
```

The `MercPubSubStore` is the class providing functionality for the actual storing of data elements.

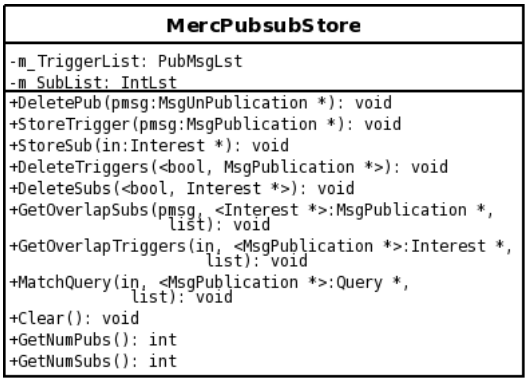

Figure 21: Mercury class: MercPubsubStore

The `StoreTrigger()` is responsible for publishing data in Mercury, and the `GetNumPubs()` is used to see how many stored elements there are at any given time. `Clear()` is used to delete all current publications and subscriptions and it uses `DeleteTriggers()` and `DeleteSubs()` correspondingly.

6.4 ANA Implementation

The compartment *Application programming interface (API)* consists of five fundamental functions which are the glue for every interaction in ANA. All compartments in ANA support some or all of this API and it allows interaction between different compartments. The API follows a publish and resolve communication model.

```
anaLabel_t anaL2_publish(const anaLabel_t compIDP, struct
    context_s *context, struct service_s *service,
    AL2Callback_t function, int separateThread, struct timespec
     *timeout);

int anaL2_unpublish(anaLabel_t compIDP, anaLabel_t label,
    struct context_s *ctxt, struct service_s *service, struct
    timespec *timeout);

anaLabel_t anaL2_resolve(anaLabel_t compIDP, struct context_s
    *context, struct service_s *service, char chanType, struct
    service_s *querierDescription, struct timespec *timeout);

int anaL2_lookup(anaLabel_t compIDP, struct context_s *context
    , struct service_s *service, struct anaL2_lkpResponse **
    result, struct service_s *querierDescription, struct
    timespec *timeout);

int anaL0_send(anaLabel_t label, void *data, int dataLen);
```

Listing 1: ANA API

- **Publish**: Requests that a certain service becomes reachable via the compartment. When successful the function returns an IDP entry point to the published service.
- **Unpublish**: Requests the removal of a certain service from the compartment. The function returns an integer value to indicate a successful removal or some error condition.
- **Resolve**: Obtains an information channel to a certain service that has been published in the compartment. When successful the function returns an IDP where the service can be reached.
- **Lookup**: Retrieves complementary description about a service.
- **Send**: Sends data to a service. The return value is an integer value to indicate a successful request or some error condition.

The two notions of context and service are meant to specify what (service) and in what scope (context) inside a compartment a certain resource will be published or resolved[3]. For example, in a file-sharing compartment one could use boolean operators like `MPEG|MKV` as context values with services like `Shawshank Redemption`. The service argument is intended as a description of the service to be published or resolved and it does not need to be interpreted by the compartment.

ANA also mandates two generic context values, namely "`*`" and "`.`". The star specifies the largest possible scope, while the dot restricts the scope to local members only. In an IP compartment this would mean `255.255.255.255` for "`*`" and `127.0.0.1` for "`.`".

In the current implementation of ANA, both the service and context arguments are represented as character strings.

6.5 MCIS Implementation

The *Multi compartment information sharing (MCIS)* functional block consists of two bricks forming the system core, in addition to other bricks using it to store and query data. Other bricks can also influence how the MCIS works internally by sending signal messages to it.

6.5.1 MCIS Bootstrap

The MCIS Boostrap brick is responsible for coordination between peers in MCIS. Because the Mercury system has a class called `BootstrapNode` written especially for this purpose, and because the responsibilities of the Bootstrap brick are limited, the source code for this brick is short and concise. The complete source code of the Bootstrap brick is shown in Appendix B to illustrate how the Mercury template works and how it can be utilized.

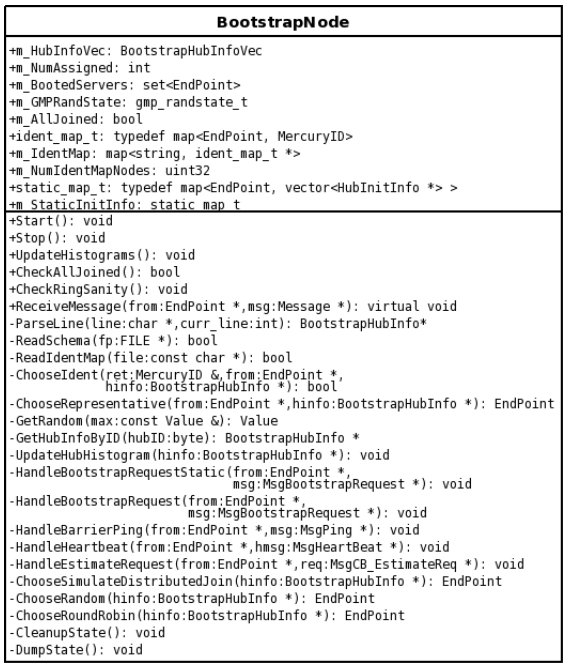

Figure 22: Mercury class: BootstrapNode

Figure 22 is representable for most Mercury classes. It seems complex, but we do not necessarily need to use all the available functions to achieve what we want.In the Bootstrap brick we use mainly a constructor and the `Start()` and `ReceiveMessage(EndPoint *from, Message *msg)` functions directly. In addition some other classes for scheduling and communication are used.

The Bootstrap brick works as the meeting point for potential MCIS peers. When it is started, the number of expected peers in the metadata compartment is passed as an argument. The brick instantiates a Mercury node, publishes itself in the IP compartment and waits for all the peers to connect to it. When all the peers have connected, the Bootstrap brick goes into a new state where it does sanity checks. This new state is implemented as an

infinite loop fetching new messages and sleeping. Peers send fixed interval heart beats to it and responsibility for the different data compartments are divided amongst the active peers.

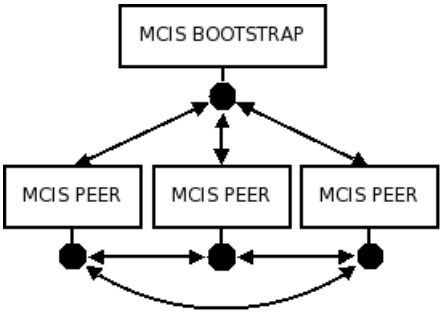

Figure 23: MCIS Peer bricks connected through the MCIS Bootstrap brick

The Bootstrap component of a standard Mercury application is optional where it is possible to make the peers connect directly to each other, eliminating the need for the Bootstrap process. This is not the case in the current version of MCIS where this is not implemented. Because Mercury has support for this form of peer discovery it is possible to extend MCIS with this feature by utilizing the corresponding functions in the Mercury API. It is something that should be considered in the future as it makes MCIS less complex. The way MCIS works now, the administrators must know in advance how many peers there will be in the system and when they all should connect.

6.5.2 MCIS Peer

The MCIS Peer brick is the core of MCIS and is responsible for all the actual work. It also utilizes Mercury classes, but the code is more comprehensive than the Bootstrap brick. The MCIS Peer brick forms ANA compartments and need to handle publish, resolve and lookup messages from other bricks.

The essential behavior of the brick is that a number of peers are started corresponding to what the boostrap brick is expecting. These peers form the metadata compartment and divide responsibility ranges for the anticipated data compartments. An example could be that two peers form the metadata compartment, where the first is responsible for data compartments which names start with a - m and the other is responsible for the remaining n - z.

The peers then make the metadata compartment available for other bricks

by publishing the keywords *mcis+compartment* in the IP compartment, or a legacy Ethernet compartment. After this publication has succeeded, any brick wanting to store or look up data can resolve MCIS and send ANA Compartment messages to it. The incoming messages to this IDP are parsed by the function `receiveFromHL()` visible in Figure 24. If it is a legitimate MCIS command it is forwarded to the corresponding handler function.

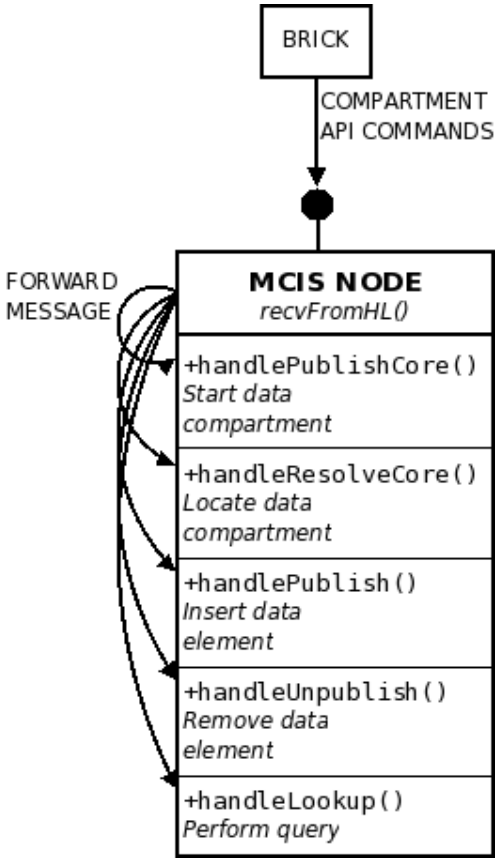

Figure 24: Message handling in MCIS

As shown in Figure 24 the accepted ANA compartment API functions are:

- Publish a datatype,
- Resolve a datatype,
- Publish data element,
- Unpublish data element and

- Lookup data element(s).

Each of these messages have an assigned function, so when a message is received and its contents gets parsed, the corresponding function is called. The first two messages are sent to the metadata compartment and the last three are sent directly to the relevant data compartment node.

An inspection of the functions one by one shows that the first function is the most complicated. It has to parse the data type, form hubs correspondingly and fork off a new MCIS peer. This new node will be responsible for the data compartment. Furthermore it has to send a message back to the original requester to notify that the publish was successful, or alternatively an error message is sent if something went wrong.

The second function is responsible for locating a data compartment. This is done by extracting the data compartment identifier and locating the corresponding IDP, either from a local array working as cache or by searching in the metadata compartment. Searching in the metadata compartment is done in the following way.

```
/* Make a MercuryID for the data compartment identifier */
MercuryID val((const char*) dataType);

/* Get exactly one data compartment, ie. MIN == MAX */
Constraint c(0, val, val);

/* Make a Query and add the Constraint to it */
Query *q = new Query();
q->AddConstraint(c);

/* Pass the Query to itself and clean up memory */
m_Router->RegisterQuery(q);
delete q, c;

/* Generate timeout values */
TimeVal v;
gettimeofday(&v, NULL);
uint64 nowtime = (v.tv_sec * USEC_IN_SEC) + v.tv_usec;
uint64 stopTime = nowtime + RESOLVE_TO * 1000000;

/* Make Event for if a data compartment is found */
MetadataEvent *e = NULL;
```

```
/* Wait for timeout or until we find the data compartment */
while(e == NULL && nowtime < stopTime) {
  e = (MetadataEvent *) m_Router->ReadEvent();

  if(e != NULL) {
    cerr << "Resolve returned: " << e << endl;
  }

  OS::SleepMillis(100);
  gettimeofday(&v, NULL);
  nowtime = ((uint64)v.tv_sec * USEC_IN_SEC) + v.tv_usec;
}
```

Listing 2: Resolving a data compartment in the metadata compartment

If no data compartment is found within the time limit, an error message is sent back to the requester. Otherwise, the IDP for the data compartment is sent.

The essentials of the publish data element function is listed below.

```
/* Extract replyto IDP from message */
replyTo = (anaLabel_t) getXRPArg(msg, XRP_CLASS_LABEL, &
    labelLen, 1);

/* Extract the actual data element from message */
dataRecord = (char *) getXRPArg(msg, XRP_CLASS_LKP_DESCR, &
    descLen, 0);

/* Create the publish event and send it */
PointEvent *ev = new PointEvent ();
anaPrint(ANA_DEBUG, "Data element '%s'\n", dataRecord);

vector<string> js;
tokenizer<comma_sep>::tokenize(js, dataRecord);

/* Go through all the attributes in the schema */
for (uint32 i = 0; i < js.size(); i++) {
  vector<string> inf;
  /* Identify lower and upper bounds */
  tokenizer<char_sep<':'> >::tokenize(inf, js[i]);
```

```
  unsigned int minVal, maxVal;
  stringstream ss((string)inf[0]);
  ss >> minVal;
  ss.clear();

  if(inf.size() > 1) {
    ss.str((string)inf[1]);
    ss >> maxVal;
    anaPrint(ANA_DEBUG, "min='%d' max='%d'\n", minVal, maxVal)
        ;
    Constraint c (i, minVal, maxVal);
    ev->AddConstraint (c);
  } else {
    anaPrint(ANA_DEBUG, "min='%d'\n", minVal);
    Constraint c (i, minVal, minVal);
    ev->AddConstraint (c);
  }

  /* Make the data item permanent */
  ev->SetLifeTime(-1);
}

/* Fetch router from local cache and send publish event*/
char *labelChar = (char *) datacompartmentIDP;
ANAMercuryNode *router = mercNodes[labelChar];

router->SendEvent(ev);
delete ev;
```

Listing 3: Publishing a data element in the data compartment

The lookup data elements handler function is very similar to Listing 2 where a query is made consisting of constraints. This query is registered with the data compartment node and the responses are sent back to the requester within a time limit. In MCIS this timeout is ten seconds. If a timeout occurs, but the results are eventually retrieved, the results are appended to future queries.

The last handler function, responsible for unpublishing data elements is almost an exact copy of the lookup function. The only differences are that the data elements are marked for deletion instead of insertion.

The source code for the MCIS Peer brick is available in Appendix B.

6.5.3 MCIS Benchmarker

The MCIS Benchmarker is an application made for testing the capabilities of MCIS. It is used for debugging, in addition to serving as the client component of the feedback control system shown in Figure 15. At runtime, the brick outputs detailed information about queries and how the MCIS responds, and results are logged to file.

Following are the different functions of the brick. Together they test all MCIS functionality and isolate problems, if any. The call sequence of the functions in Listing 4 is essential because they depend on each other.

```
/* Resolve the IP brick */
int test_ipbrick();

/* Resolve the MCIS peers forming the metadata compartment */
int test_mcis(char *anaip);

/* Start a new data compartment.
 * A MCIS peer forks off a new Mercury
 * The new node will comply to the specified schema
 */
int test_publish_schema(char *schema);

/* Resolve the newly created data compartment */
int test_data_compartment(char *schema);

/* Send a data item as comma separated list of values
 * The data comparment will store the item
 */
int test_save_data(char *filename);

/* Send queries on the form VAR1:MIN:MAX,VAR2:MIN:MAX
 * Unspecified variables are considered to be wildcards
 */
int test_query(char *query);
```

Listing 4: MCIS benchmark functions

The logs that the MCIS Benchmarker provide can be used to analyze the effects of the resource adaptation. Different levels of information can be specified, ranging from `ANA_NONE` with no output, to `ANA_DEBUG` with a lot of

information. By using a level in between called `ANA_NOTICE` for the queries and their responses, it is easy to separate this from the irrelevant output such as debug information.

Service	**ANA API function**	**Where**
ip	anaL2_resolve	NODE_LABEL
MCIS_1	anaL2_resolve	IP compartment
Data cmp id	anaL2_publish	Meta data compartment
Data cmp id	anaL2_resolve	Meta data compartment
data_item	anaL2_publish	Data compartment
query	anaL2_lookup	Data compartment

Table 2: ANA API functions used in the MCIS Benchmark brick

The services in italic are substituted for actual values in the MCIS Benchmark brick. A query string could for instance be `‘‘0:10:100’’` meaning a query for all values between 10 and 100 in attribute 0.

The complete source code for the MCIS Benchmark brick is available in Appendix B.

6.6 Resource adaptation

The resource adaptation bricks are created to provide self-optimization for the MCIS FB. The two bricks have distinct responsibilities, but work together.

6.6.1 System monitor

The System monitor does periodic inspection of the system resource utilization according to Table 3. The structure is populated by the Linux kernel and is equivalent to obtaining system statistics from the /proc pseudo file system [1]. The brick logs this information together with a timestamp for later inspection while still being available through a published IDP for other bricks wanting real-time data. In our experiments the System monitor operates as the measurement component of the feedback control system shown in Figure 15.

long uptime	Seconds since boot
unsigned long loads[3]	1, 5, and 15 minute load averages
unsigned long totalram	Total usable main memory size
unsigned long freeram	Available memory size
unsigned long sharedram	Amount of shared memory
unsigned long bufferram	Memory used by buffers
unsigned long totalswap	Total swap space size
unsigned long freeswap	Swap space still available
unsigned short procs	Number of current processes
unsigned long totalhigh	Total high memory size
unsigned long freehigh	Available high memory size
unsigned int mem_unit	Memory unit size in bytes
char _f[20-2*sizeof(long)-sizeof(int)]	Padding for libc5

Table 3: Overall system statistics structure (sysinfo.h)

The implementation of the System monitor consists of two functions in addition to the standard ANA template. It has one timer based function doing periodic inspections, and one callback function for answering requests from other bricks. The IDP of this callback function is visible as node_monitor in Table 4. The service in italic is substituted for an actual value at runtime. This could be for example decision_maker.

Service	ANA API function	Where
node_monitor	anaL2_publish	NODE_LABEL
requester	anaL2_resolve	NODE_LABEL

Table 4: ANA API functions used in the System monitor brick

The complete source code for the System monitor brick is available in Appendix B.

6.6.2 Decision maker

The Decision maker is a link between the System monitor and a MCIS Peer. It is responsible for telling the MCIS Peer to start or shut down an attribute hub based on data retrieved from the System monitor. The signaling is triggered when resource consumption parameters from the System monitor reaches predefined thresholds. If resource consumption is above an upper threshold a signal is sent to MCIS asking it to shut down an attribute hub, and the opposite is true if resource consumption goes below a lower threshold.

Service	ANA API function	Where
node_monitor	anaL2_resolve	NODE_LABEL
decision_maker	anaL2_publish	NODE_LABEL

Table 5: ANA API functions used in the Decision maker brick

The MCIS Peer is not resolved in the Decision maker because signaling between these two bricks is done through POSIX-style signals, a simpler form of inter-process communication. This is also the reason that no MCIS Peer IDP is visible in Figure 25.

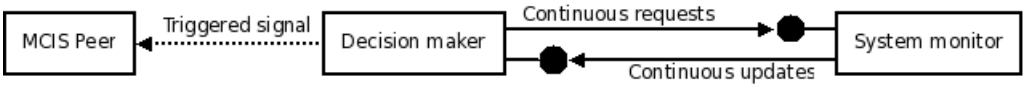

Figure 25: Decision maker communication

The main parameter of the decision making algorithm is system load. System load is a measure of the amount of work that a computer has to do. On Linux systems the load is based on the CPU queue length which is a list of all the processes that want to do computational work. This means that the load increases when several processes wants the CPU at the same time.

An idle computer has a load of 0, and each process that is waiting for CPU time, i.e. not in a sleep state or waiting for I/O, adds to the load by 1. Research suggest that decisions based on CPU queue length does better for load balancing compared to CPU utilization[7].

Figure 26: CPU queue

Figure 26 shows a CPU queue with two processors wanting to use the CPU at the same time. This means that the system load at that exact moment is 2. However, this load number will fluctuate from one second to another depending on the running processes. It is therefore important to use an average value and not make decisions based on a load number that may be obsolete or is just a temporary spike.

The Linux kernel offers average values of 1, 5 and 15 minutes as seen in Table 3. These load numbers are calculated as the exponentially weighted moving average within the window, meaning that all the recent load numbers are regarded, but that the newest are most relevant.

On single CPU systems, where only one process can actually be running at a time, the load average is the same as a percentage of utilization. For example, a load average of 1.86 means that the CPU was overloaded by 86%. One process is running, while 0.86 processes have to wait. The load will only increase above 1 if the CPU can not service all requests. This is important to know when deciding thresholds for the feedback control system.

The system load is the predominant parameter for the Decision maker. It uses the load gathered from the one minute window as its trigger because it will make the decisions faster than if it were based on one of the other two averages. The five and fifteen minute averages will not reflect changes fast enough to be useful for resource adaptation. A load number of 1 is used as the upper threshold because a higher load number means that processes are queued and have delayed execution. The complete source code for the Decision maker is available in Appendix B.

6.7 IP functional block

The IP functional block is an implementation of the classic network stack in ANA. It provides functionality for framing, checksum computation, addressing, forwarding and more separated into four bricks.

- **ip_enc**: IP encapsulation, checksumming and address resolving.
- **ip_fwd**: IP forwarding table.
- **ip_cfg**: Configuration program similar to Linux' ifconfig.
- **rip**: Routing Information Protocol (RIP) implementation.

A brick can make a service reachable through the IP compartment by using the standard ANA API publish function, and because the Mercury system is using endpoints, the different MCIS peers use the IP functional block to communicate with each other. The context parameter is used to define who the service is available for, where the * context means that the service can be resolved from anywhere and the . context means that only the node-local scope can resolve it[13]. The complete source code for the IP bricks is available at the ANA project webpages at *http://www.ana-project.org/* and on the attached DVD.

6.8 IMF and the Video-on-Demand functional block

Some of the possible uses for MCIS is in the mentioned monitoring framework and in content distribution. The *Video-on-Demand (VoD)* functional block is a streaming service where the content is retrieved from several nodes simultaneously. It has built-in functionality for search and a way of organizing the nodes that have the data based on delay. If a node searches for a video, a list of nodes with the corresponding content is returned. The node can then start downloading different parts from different nodes, ending up with the complete video. The parts are simultaneously made available to other nodes when retrieved. The VoD FB needs a place to store information about video titles, file sizes, blocks and corresponding peers and the MCIS could be used for this.

Monitoring in ANA is implemented as a compartment called the *Integrated Monitoring Framework (IMF)* and when another compartment requires monitoring information for its decision processes, the IMF initiates and delegates the data collection and distribution of results. The IMF provides functionality to gather, combine and exchange information between any participant at any time. The complete source code for the IMF and VoD bricks are available at the ANA project webpages at *http://www.ana-project.org/* and on the attached DVD.

7 Evaluation

The service we evaluate in the following sections is a distributed information sharing system called MCIS and a self-optimization scheme for it. The first thing we need to do in order to evaluate this service is to determine the right goals, measures, environment and evaluation techniques. We strive to keep this environment simple and easy to understand.

7.1 Goals and outcomes

We know that MCIS is an important component in ANA. There are many different applications that can benefit from using MCIS to store and query data, and they have high demands for performance that MCIS must comply to. They expect correct results, high efficiency and fast responses for their queries. In addition, ANA mandates the use of self-star attributes such as self-optimization for all its components.

The objective of our evaluation is therefore to investigate the differences between MCIS with and without self-optimization and show that resource adaptation is possible. If resource adaptation is possible, we also want to investigate if MCIS can perform better when adapting to system resource consumption in form of CPU load. The desirable outcome of our evaluation is higher efficiency and an increased number of successful query results when resource adaptation is applied. If we achieve this goal, we have successfully designed a self-optimizing, distributed information sharing system for the future Internet.

7.2 Metrics

There are several possible criteria to inspect for our evaluation: response time, throughput, reliability and availability. Mercury does explicit load balancing and replication of data, so reliability and availability is expected to be good when several MCIS nodes cooperate, even without our modifications. The demands that applications using MCIS have are primarily:

- How many queries can MCIS handle simultaneously without failing, and

- how much time does it take for an application to perform a query and receive correct results back.

The number of simultaneous queries that an interactive system like MCIS can handle is known as throughput rate and is expressed as queries per minute. Initially, throughput increases as more concurrent queries are run, but after a certain rate is reached, the throughput stops increasing and may even start to decrease[10]. This is because the system can get overloaded and become unable to answer incoming queries, i.e. the overhead of handling incoming queries affect the ability to answer and send outgoing replies. High throughput is very important in content sharing applications where huge amounts of data need to be distributed.

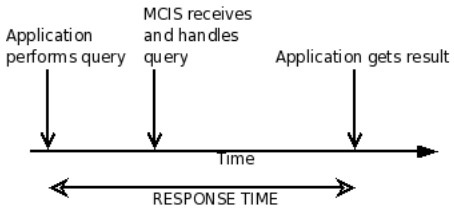

Figure 27: Response time

Response time is defined as the interval between when an application performs a query and when it gets a corresponding response back, visualized in Figure 27. As with throughput, the response time generally increases when more simultaneous queries are made[10]. Low response time is very important in applications where data have a tendency to become outdated quickly, like in real-time monitoring.

We choose to evaluate our system based on *throughput* and *response time* as these metrics reflect the demands and requirements that applications using MCIS have for performance.

7.3 Parameters and parameter values

A parameter is defined as a system characteristic that affect the performance of the evaluated service[10]. There are many parameters that can affect the performance of MCIS. Some of these are related to hardware and others to the running software. When doing repeated evaluations we strive to eliminate differences between each run and keep the influencing parameters to a

minimum. We want to fully control the remaining parameters and vary them to see the impact it has on the performance.

One parameter in our evaluation is the machines we use and the MCIS nodes they run. The machines are *Intel Pentium 4s* with *2.60 GHz* and *1 GB of memory*. They are all running Ubuntu GNU/Linux with a 2.6.31-14 kernel. We use the same configuration on all machines to ensure that queries are treated with the same processing power every time, independently of which MCIS node they are forwarded to. All the machines are connected directly to each other through the same switch. There will be two MCIS nodes and one client node running as shown in Figure 28.

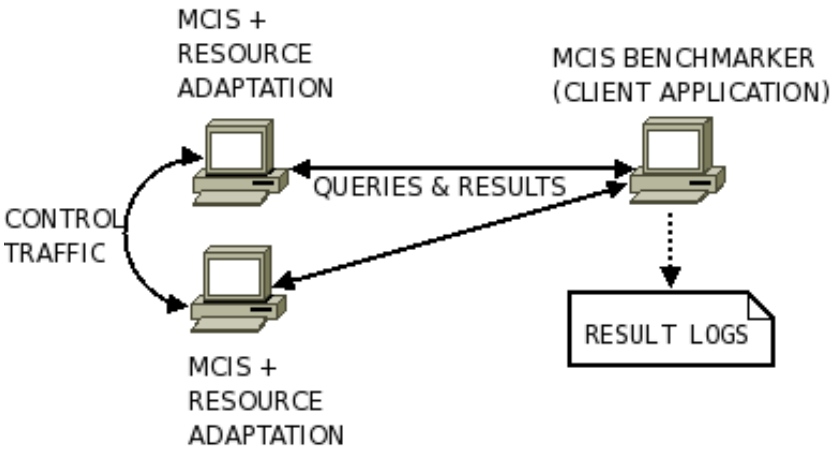

Figure 28: Three-machine setup

Another parameter is the software the machines are running during the evaluation. We use a standard GNOME desktop without unnecessary applications started, meaning that only the ANA Core, MCIS, MCIS Benchmarker and the resource adaptation bricks are executed. The only overhead we have regarding software is the operating system.

The number of hubs in each MCIS data compartment is one of the key parameter in our studies. It will vary automatically according to resource consumption and not be modified by us directly. The implications of the hub count is very significant and will be investigated thoroughly below.

External load is also an important parameter in our evaluation. It is the additional amount of processing that is delivered to the machines where MCIS run causing the resource adaptation to function. We use synthetic load to have total control over the load span and when it is applied. The external

load is generated by two programs with the purpose of using a predetermined amount of CPU.

generate_cpu is a program with a greedy while-loop using as much CPU as possible, and cpulimit is a program that limits the CPU usage of a process, expressed in percentage. cpulimit does not act on the process' nice value or scheduling priority, but on the real CPU usage[9]. Together we use these two programs to control exact CPU utilization at all times. For example, the commands for using 100% CPU is as follows:

```
generate_cpu &; sudo cpulimit -e generate_cpu -l 100
```

The parameters we vary in our studies are the *number of inserted data elements (DE)* and the *number of queries per minute (QPM)*. We choose these parameters because they correspond well to what the load of MCIS will consist of. By changing one or both of these parameters we can investigate the implications and understand in which situations, if any, resource adaptation can improve MCIS.

7.3.1 Data structure and elements

The data we use in our evaluation are real Internet traffic traces gathered on March 31, 2009, courtesy of CAIDA[17]. This data is chosen because it represents real world data for MCIS. We insert several thousand of these data elements, but vary the exact number to determine the affect this has on the resource adaptation. We have previously tested MCIS with even more inserted data elements, but this makes MCIS unstable.

ATTRIBUTE	MINIMUM VALUE	MAXIMUM VALUE
Source IP	0	4294967295
Source port	0	65536
Destination IP	0	4294967295
Destination port	0	65536
Bytes	0	1600000000

Table 6: Schema for evaluation

Table 6 shows the schema, attributes and value ranges used as parameters in our evaluation. MCIS does not support dots in any of its data types, so the IP addresses in the CAIDA traces are converted to long with the IP2Long program available in Appendix C.

7.3.2 Query

We aim at keeping our evaluation realistic and want to choose queries that represent real world usecases. Given our schema, a typical investigation on Internet traces is to locate malicious traffic and vulnerable clients.

Internet worms are known to open backdoors on certain ports for their developers to be able to make patches and perform attacks. If different versions of a worm are opening ports ranging from 52830 to 52839 an equivalent MCIS query to find this hole is `1:52830:52839` on the given schema.

We have chosen to use this query because it is realistic while also utilizing Mercury's ability to perform range-based queries. It is invoking route calculations and might fail if the CPU load on the machine running MCIS becomes too high. We query MCIS between 100 and 300 times per minute because this is a good range of what MCIS is able to handle in its current version. We have previously tested MCIS with even more concurrent queries, but this makes MCIS unstable. To ensure that correct results are delivered, we personally inspect the CAIDA traces to find the expected number of results.

7.3.3 Trigger thresholds

Our resource adaptation scheme is based on a feedback control system which signals MCIS when resource consumption, i.e. CPU load, reaches certain levels. We need to determine trigger thresholds for the load that will invoke the resource adaptation scheme, and tell MCIS to either shut down an attribute hub or start a new one.

Processor usage is the amount of time an application uses for processing instructions in the *central processing unit (CPU)*. In MCIS this means performing queries, and other computational work related to data ranges. This implies that an overloaded system might not answer queries when the CPU is busy doing calculations for other applications, as explained in Section 6.6.2. The consequence of this fact is that the upper threshold should be 100%. More than 100% load is possible, but this means that the system is overloaded. The lower treshold can be any number below MCIS' own requirements for CPU. For example, if MCIS needs 5% CPU to function properly, the lower treshold can be 95%.

7.4 Evaluation technique

There are several ways to do performance evaluation. Three common ways are analytical modeling, simulations and measurements[10]. They are equally good, but designed for different purposes. The best performance evaluations consist of a combination of all three techniques, but time and resources seldom allow a combination of all three. This is true for our work, so we have to choose one of these evaluation techniques.

Analytical modeling is often done by calculations and theoretical proofs. This is a good evaluation technique for several applications, but is not the best approach for our evaluation. The reason is that internal structures of Mercury and MCIS are too complex to represent mathematically, and constructing a reasonable function would be almost impossible. In addition, the results of an analytical model can seldom be trusted until it is validated by a simulation model or measurements. Simulations are closer to reality, but still not precise. Details are omitted, and several assumptions must be made. Measurements are real system investigations. This technique is best for our evaluation of MCIS because we have a complete, running system ready to be tested. All the different parts of our information sharing system and the self-optimization extension are in place and work together.

We use a three-machine-setup for our evaluation. Two machines run MCIS and the resource adaptation bricks, and the last machine run the MCIS Benchmarker application, in charge of storing and querying data. The varying factors, *inserted data elements* and *queries per minute*, are changed in the MCIS Benchmarker. After the experiments we examine the result logs with respect to the chosen metrics.

7.5 Experiments

We conduct our evaluation experiments in two phases with one local and one distributed test. In the first test we use one MCIS node and experiment with a wide range of parameters. This test is an attempt to narrow our parameter values and prepare for the second, distributed test. It will establish that MCIS works and help us determine the CPU requirements it has. In the second test we setup a distributed system with two collaborating MCIS nodes and adjust the parameters corresponding to what we learned in the first test.

Figure 29 shows the entire setup for our experiments in the distributed test.

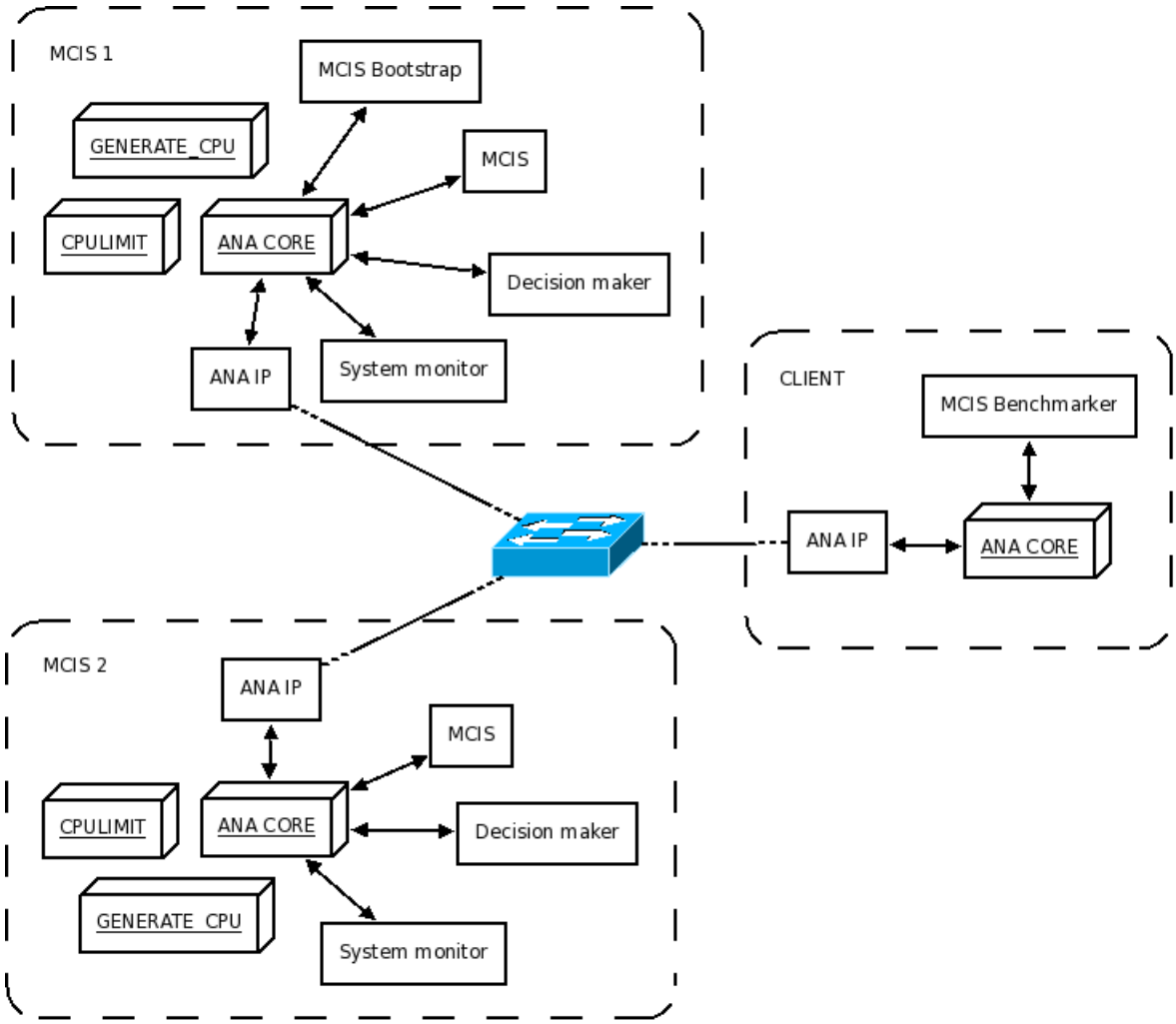

Figure 29: Complete experiment setup

It shows how the three nodes communicate through the ANA IP and also which programs and bricks that run on the different machines. This figure is a detailed equivalent of Figure 28 that show the internal components.

7.5.1 Local test

The first evaluation test is performed on one MCIS node to investigate how much load MCIS can handle and what our parameter values should be. We experiment with a wide range of parameters values and try to identify the impact different variations have on the query results.

To determine how much CPU MCIS uses we do a test without introducing the external, synthetic load, and without starting the resource adaptation bricks. When MCIS is the only running application, the CPU utilization during that run indicates how much CPU MCIS uses. After these initial tests we investigate the results and decide our final parameters.

DE	QPM	AVERAGE LOAD	SUCCESSFUL QUERIES
3000	300	0%	0%
3000	200	5%	0%
3000	100	5%	0%
2000	300	4%	0%
2000	200	4%	0%
2000	100	2%	84%
1000	300	4%	31%
1000	200	3%	47%
1000	100	2%	93%

Table 7: First evaluation test

The first thing we see in the results in the local test, shown in Table 7, is that MCIS is not using much CPU resources, even when many data items are inserted and many queries are made concurrently. The average system CPU load during the experiments is between 0% and 5% and because MCIS is the only running application, it is accountable for most of this utilization. The low CPU usage implies that the external, syntethic CPU load should be above 95% in order to interrupt MCIS, and it also confirms that our upper, resource adaptation trigger treshold should be 100%.

Figure 30 shows that if an external CPU load of 70% is introduced, MCIS will still have enough resources available because it only requires a maximum

Figure 30: CPU consumption

of 5% CPU.

The low CPU load in the run with 3000 DE and 300 QPM is a consequence of the operating system running out of threads. In order to assist different clients simultaneously in a non-blocking fashion, one thread is created for each incoming query. When too many queries are made concurrently, the operating system is unable to create enough threads. This causes the machine to become unstable after a short period of time, and MCIS is unable to do any meaningful work.

We can also see from the local test that one MCIS node is uncapable of handling more than 100 queries per minute when it has stored 2000 data elements or more. When the queries are distributed among two nodes these results might improve, but our further investigations are focused on 2000 inserted data elements and below.

	MINIMUM	**MAXIMUM**
Data elements	1000	2000
Queries per minute	100	300
External CPU load	100%	100%
Trigger thresholds	95%	100%

Table 8: Refined parameter values

Table 8 shows our final parameter value ranges. We want to perform one experiment for each of the varying query rates and number of inserted data elements, without resource adaptation enabled, and similar experiments with resource adaptation enabled.

7.5.2 Distributed test

In the distributed test we perform eighteen experiments with varying parameters. This test is meant to demonstrate a realistic use of MCIS and we choose a specific workload to reflect this. The experiments are divided into three categories based on the number of inserted data elements and corresponding graphs are created to visualize the results.

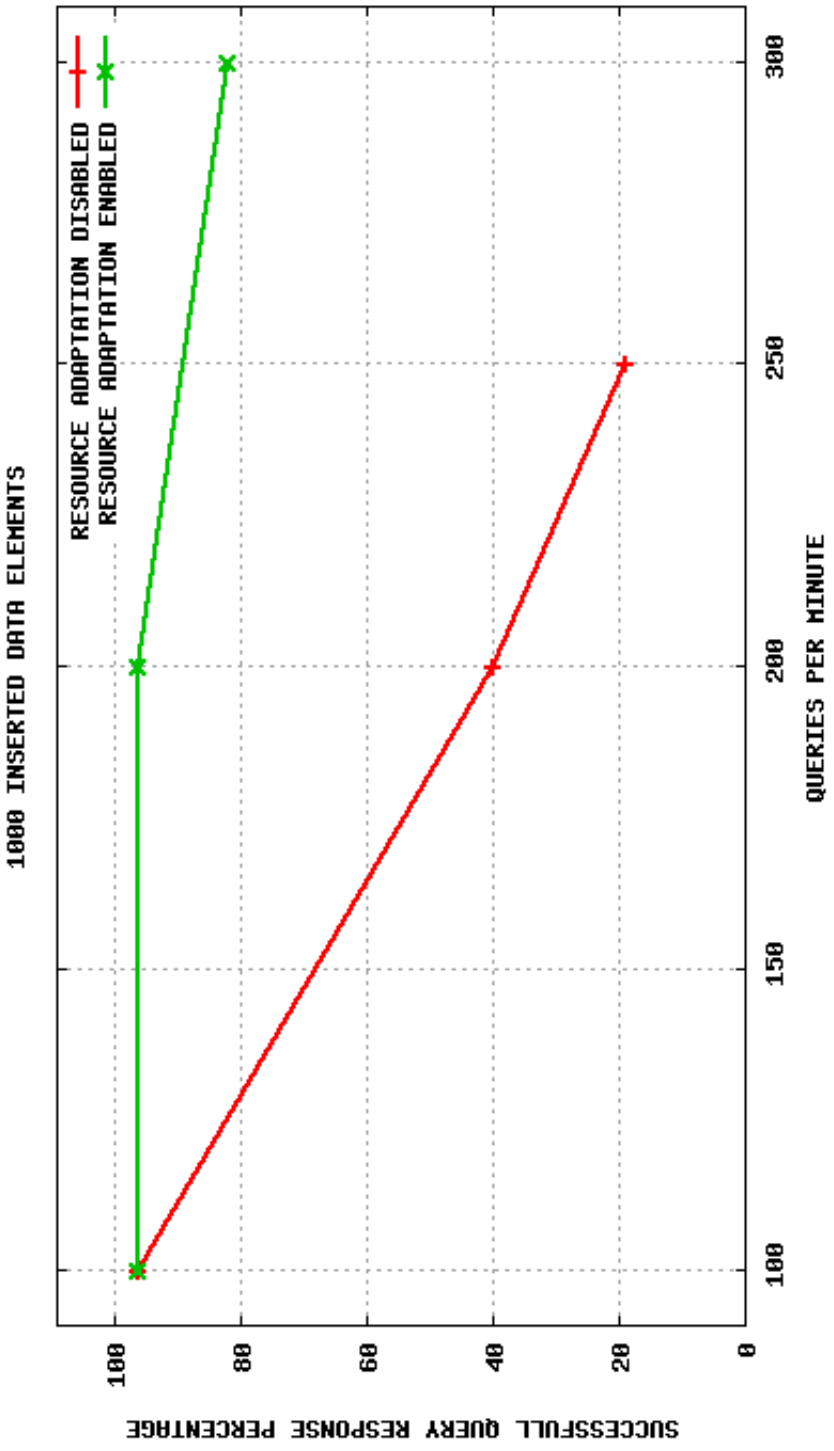

Figure 31: Query rate vs. Success rate (1000 data elements)

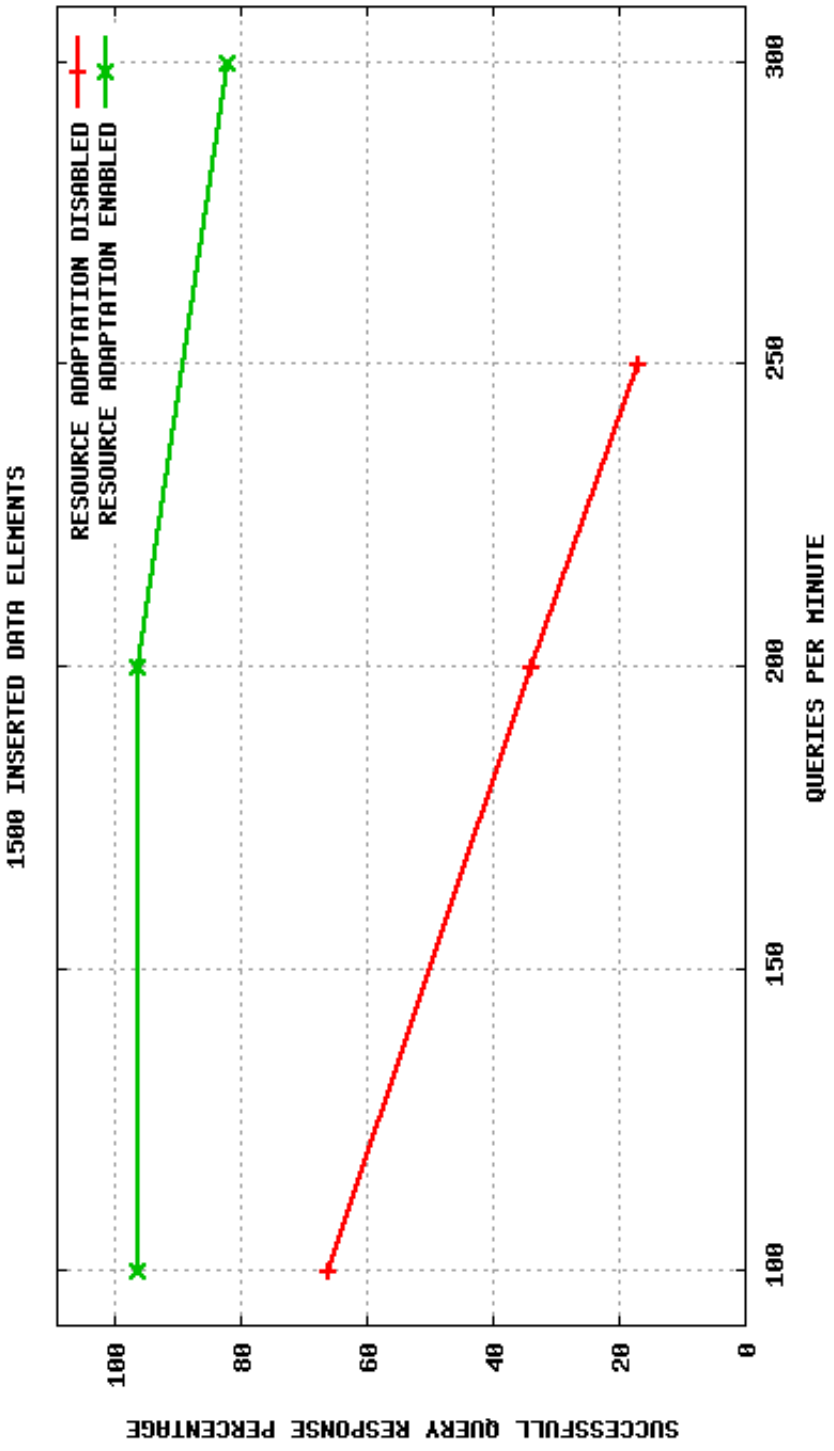

Figure 32: Query rate vs. Success rate (1500 data elements)

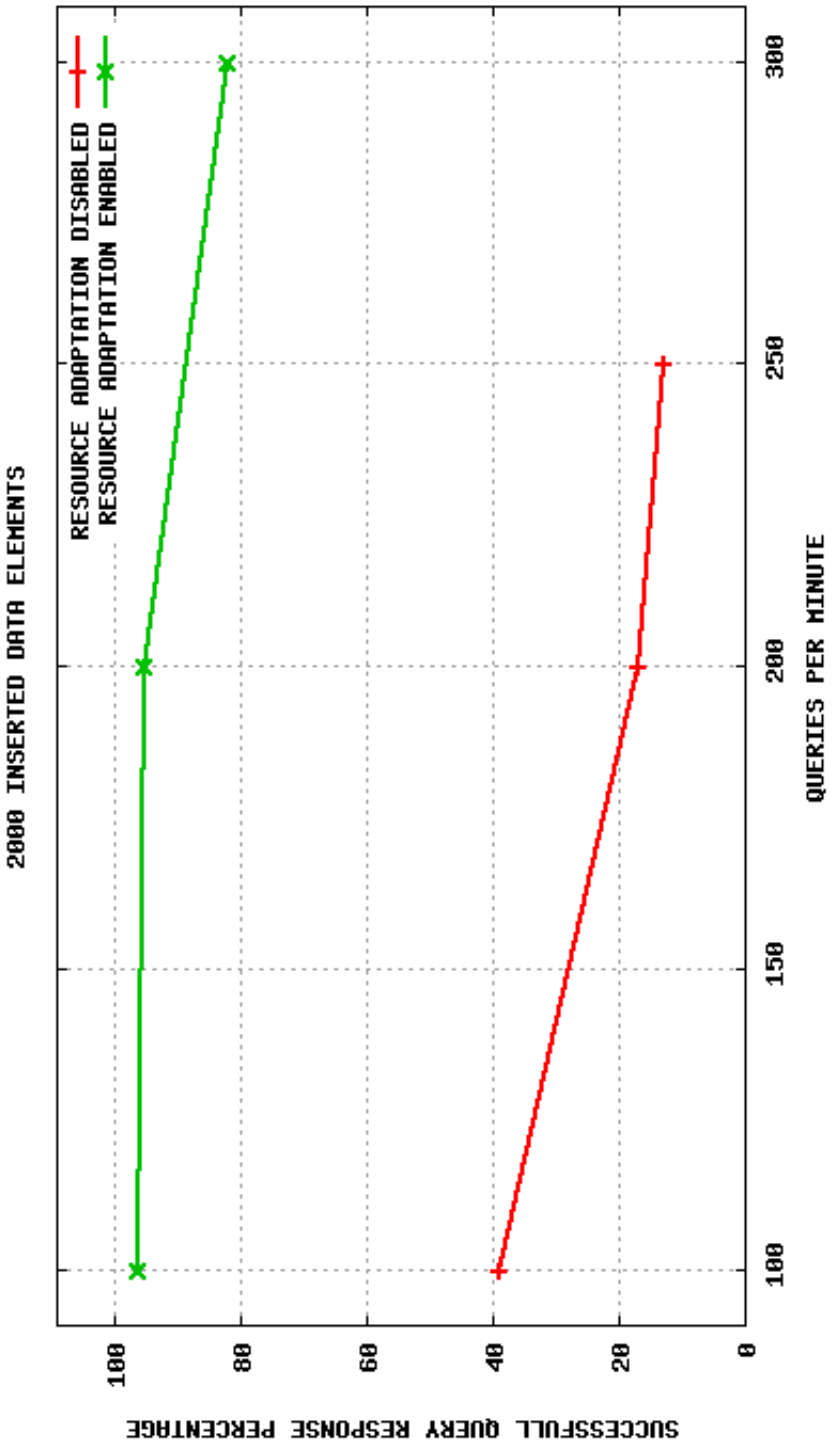

Figure 33: Query rate vs. Success rate (2000 data elements)

We are unable to achieve consistent results when trying to query MCIS 300 times per minute, so the query rate is decreased to 250 when resource adaptation is disabled.

7.6 Analysis of results

The purpose of our performance evaluation is to test how good the self-optimization scheme works in respect to our chosen metrics, throughput and response time. Both of these metrics are important when evaluating the performance of MCIS and determining the impact of the self-optimization. We divide our analysis into two sections, one for each of the metrics respectively.

7.6.1 Throughput

Throughput in MCIS is the number of successful queries delivered to the client application, MCIS Benchmarker in our evaluation. It represents the performance of the system, and when comparing throughput between the experiments with and without resource adaptation it can indicate how good the self-optimization works.

Figures 31, 32 and 33 show clearly that more queries are answered successfully, and that the throughput is higher, when self-optimization is enabled. This is the case for all our experiments, except when the number of inserted data elements is 1000 and only 100 queries are made per minute. In this specific case the resource adaptation has no positive effect on throughput. This is most likely because MCIS does not have enough incoming queries to be interrupted by the external CPU load in either case.

When resource adaptation is enabled MCIS is able to sustain close to 100% success rate for both 100 and 200 queries per minute regardless of how many data elements are stored. This is not the case for when adaptation is disabled. In the experiments where resource adaptation is disabled, the success percentage drops drastically when the query rate is increased from 100 queries per minute to 200 queries per minute. This is especially true for the experiments with 1000 and 1500 stored data elements, visible in Figures 32 and 33.

Our analysis shows that resource adaptation can have a positive effect on throughput. Most improvement is achieved when the number of stored data

elements is large or when query rate is high. Conversely, resource adaptation does not seem to impact throughput when both the query rate is low and the number of stored elements is small.

7.6.2 Response time

Response time is an important part of the performance of MCIS, and it is vital that the resource adaptation does not come at the expense of it. If MCIS uses more time to answer queries when resource adaptation is enabled, the increased throughput is less valuable.

Response times have a tendency to suffer at the highest throughput rates[10]. Because of this fact we choose to only analyze response times in the cases where throughput is at its highest, i.e. when most successfull queries are made per unit of time. We measure the average response times of 100 successful queries when 1000, 1500 and 2000 data elements are stored and 4 concurrent queries are made per second, equivalent to 250 queries per minute.

Because of a design flaw in the current version of MCIS we have not been able to test response times in the way we hoped. In its current version, MCIS uses an internal timeout of 10 seconds before it returns the collected data elements to the client. This means that the response time of a query can never be lower than this. The way this should have been done is in a callback-style fashion where partial results would be forwarded to the client as they were discovered by the MCIS. This way, the response times could be measured correctly and would never be more than 10 seconds. They way MCIS works today, were data elements are collected until a timeout is reached and then sent together, response times will always be close to this timeout.

An ideal result of our unoptimal experiments is response times close to the internal timeout in MCIS. A nonce is used for each query to determine exactly when it answered and this information is logged to file using standard ANA logging mechanisms with a precision of one second. The average response time is 10 seconds in all our experiments as expected.

These results are as good as they can be because of the timeout limitation in MCIS. Because these tests are performed when MCIS has the biggest load, both in form of query rate and external CPU load, we have strong reason to believe that the same results would appear for lower query rates and external CPU loads as well.

8 Conclusions

8.1 Summary of contributions

The main goal of this work was to develop MCIS, a distributed information sharing system for an autonomic network architecture. We have successfully accomplished this goal by modifying an existing software package called Mercury in accordance with the principles of the Autonomic Network Architecture Project. The MCIS consists of two bricks, one bootstrap brick and one peer brick, working together to form ANA compartments responsible for distributed information sharing.

We have also been able to develop a self-optimization extension for MCIS, consisting of two bricks responsible for measuring and analyzing resource consumption. The System monitor gathers information about the system it is running on and the Decision maker informs MCIS when resources reach certain levels. These two bricks interact with each other and with MCIS in order to provide a resource adaptation scheme with the objective of increasing throughput.

The components we have made have been evaluated with a small scale experiment. We configured a system consisting of two MCIS nodes and one client and were able to administer both local and distributed tests in order to investigate how good the MCIS works and the impact of the self-optimization extension. This evaluation showed that MCIS works and scales well when several nodes work together. It also showed that self-optimization can be done through resource adaptation and that an increase in throughput can be expected when using it.

We have seen that the resource adaptation is very effective when the number of stored data elements in MCIS is large or many queries are made to it simultaneously. The resource adaptation rendered an improvement in query success rate from 39% to 96% when 2000 data elements were stored and 100 concurrent queries were made per minute, and from 19% to approximately 90% when 1000 data elements were stored and 250 concurrent queries were made per minute. The success rate increased from 17% to 95% when 200 queries were made per minute and 2000 data elements were stored, and similar improvements were achieved in almost all our experiments.

With resource adaptation, MCIS was capable of sustaining close to 100%

success rate with 200 queries per minute or less, regardless of how many data items were stored. With 300 queries per minute, the success rate dropped to 82%. When resource adaptation was disabled, the success rate decreased rapidly when moving above 100 queries per minute and we were unable to get consistent results when trying to query MCIS more than 250 times per minute.

8.2 Critical assessment

Even though Mercury is designed to be a modular and generic system template, we encountered several problems when developing MCIS. Some of them were bigger than others, but none of them where so grand that we did not manage to solve them.

The first problem we had was trying to get an overview of both the ANA core and the Mercury code. These are both very complex systems, developed by several individuals over the course of many years. Mercury alone has over 32 000 lines of code[18] and several contradicting programming styles. A programming style is a set of guidelines used when writing source code and is claimed to help programmers to better understand source code. Good style can be said to be a subjective matter, but a mixture of different styles and implementation techniques is prone to generate questions and confusion. After working with the code for some time this problem slowly faded.

The second, much bigger problem we struggled with, was making MCIS available to the other partners in the ANA project. We have been working on a machine with older compiler and library versions and when others tried to compile Mercury, it failed. We solved this the hard way by thoroughly going through each error, correcting and re-writing, so that Mercury and MCIS now compile on newer configurations. This was very time consuming, but a necessity if MCIS was going to be used by other ANA partners.

Another issue we have struggled with throughout this work is the instabilities of Mercury. Mercury is well designed with many clever features and is relatively manageable to work with, but because it is not maintained by the original developers it has some bugs and shortcoming. If something out of the ordinary happens, it is not likely to endure it. Malformed requests can lead to segmentation faults and unexpected behavior is known to happen from time to time.

We have also discovered a design problem in MCIS in how queries are handled. MCIS collects data elements for several seconds until a timeout is reached, instead of sending partial results when they are discovered. This makes query responses slow and impossible to measure in a sensible way. Unfortunately we have noticed this too late in the process of this thesis and do not have enough time to correct it and perform meaningful response time measurements.

Lastly, the ANA core developers changed the ANA API midway in the course of this thesis. The MinMex continued to function as before, but all the API calls had to be re-written according to Table 1. New parameters where introduced and wrapper structures had to be added for each message sent. These changes did not affect the entire Mercury system, only ANA specific parts, but was still difficult and time consuming to finish.

If we were to do this thesis again and encounter the same types of problems, we would use the experience and knowledge of our colleagues, both locally at the department and at the other research institutes, to a much larger degree than now. We often struggled too long with individual problems, especially in the beginning when we did not have the complete overview. A subtle hint or a point in the right direction is often enough to advance further when working with such complex systems. We learned a lot in this process, but it took too much time away from the actual work.

8.3 Future work

There are many extensions to MCIS and the self-optimization scheme that could be made. Some of them would be more time consuming than others to implement and it is therefore natural to categorize the future work into two categories, namely short- and long-term. Short-term is considered to be anything less than a couple of months where as long-term tasks would require several months of work.

8.3.1 Short-term goals

As mentioned in Section 6.5.1 the Bootstrap brick is mandatory in MCIS, but not in all Mercury applications. The use of a Bootstrap brick limits how flexible the MCIS is because the metadata compartment must be setup in advance with how many peers it expects to participate. When we know that

peers have a tendency to fail we understand that the metadata compartment could potentially disappear after some time. This is not as stable as we would like and is therefore not a permanent solution. It would be much better if users could start an ANA node, connect to an existing peer and be part of the metadata compartment whenever they want to. This is possible by extending the MCIS Peer brick.

There are also many improvements and extensions that could be made to the System monitor brick and the Decision maker brick. One of these improvements is to switch from POSIX signals to the standard ANA API for communication between the Decision maker and the MCIS Peer. We chose the former method because it was an easy way to implement and test the resource adaptation scheme, but this approach uses legacy system-calls and is not in accordance with ANA philosophy.

Another extension we would like to see in the future is making the decision maker more intelligent. Today it signals the MCIS solely based on predetermined thresholds, but this could change in the future. The brick could for example aggregate data over time or compare several different metrics in order to make its decisions. Similarly, the System monitor brick could be extended to inspect many other elements of the running system, besides processor and memory consumption. What in particular to investigate depends on the context and what the users want, but we imagine this brick to continually grow when new needs arise.

Lastly, it would be interesting to see the effect of having more than two MCIS nodes collaborating when doing the distributed evaluation. We saw a big increase in the number of successful queries when moving from one to two nodes, and it would be nice to introduce even more nodes to see the impact this has on performance. MCIS scales very well and its full potential is not shown in our tests. Unfortunately, we do not have access to more machines or enough time to run additional performance evaluations, but given these resources we expect the number of successful concurrent queries to advance beyond what we have seen so far.

8.3.2 Long-term goals

To make MCIS feature complete, and make the data compartments support all the expected ANA compartment API functions, the unpublish function must be implemented in the metadata compartment. As opposed to publish

in the metadata compartment, the unpublish function is supposed to stop a data compartment. This means to gracefully shut down all the forked off Mercury instances responsible for a certain data type and remove all the stored data. This requires a new global event in Mercury and coordination between all the participating peers. The data compartment peers must also be removed from the metadata compartment. Mercury has functionality to stop nodes, but because of the complex coordination between peers and the different compartments, this is not a trivial task.

Another feature that MCIS would gain benefit from is subscriptions. A subscription is a form of continued interest in certain data types. It would be a good feature for applications using MCIS to be able to report their interest in certain data types and get updates whenever changes are made to them. For example, a peer-to-peer content distribution application could subscribe to a data type responsible for different parts of a file. In this way it will always be aware of which peers have which parts of the file. Mercury has classes for these types of situations in its API, and ANA encourage the use of subscriptions, but we have not had the time to look into how to extend MCIS to support this.

REFERENCES

[1] *Linux Programmer's Manual*, August 1997. SYSINFO(2).

[2] ANA Project. *ANA Blueprint - First Version Updated*, sixth framework programme edition, February 2008.

[3] ANA Project. *ANA Core Documentation*, sixth framework programme edition, December 2008. Deliverable D.1.11.

[4] Ashwin Bharambe, Jeffrey Pang, and Srinivasan Seshan. Colyseus: A distributed architecture for online multiplayer games. *ACM/USENIX NSDI*, 2006.

[5] Ashwin R. Bharambe, Mukesh Agrawal, and Srinivasan Seshan. Mercury: Supporting scalable multi-attribute range queries. *ACM SIGCOMM*, 2004.

[6] Francois Cantin, Vera Goebel, Bamba Gueye, Dali Kaafar, Guy Leduc, Matti Siekkinen, Jin Xiao, and Maxwell Young. Deliverable d.3.8 - self-optimization mechanisms. January 2009.

[7] Domenico Ferrari and Songnian Zhou. An empirical investigation of load indices for load balancing applications. Technical Report UCB/CSD-87-353, EECS Department, University of California, Berkeley, May 1987.

[8] Vera Goebel, Bamba Gueye, Theus Hossmann, Guy Leduc, Sylvain Martin, Christoph Mertz, Ellen Munthe-Kaas, Thomas Plagemann, Matti Siekkinen, and Dorota Witaszek. Deliverable d.3.7 - integrated monitoring support in ana. February 2009.

[9] Gregor Herrmann. *Linux Programmer's Manual*, November 2006. CPULIMIT(1).

[10] Raj K. Jain. *The Art of Computer Systems Performance Analysis*. Wiley, April 1991.

[11] Christophe Jelger, Christian Tschudin, Stefan Schmid, and Guy Leduc. Basic abstractions for an autonomic network architecture. *World of Wireless, Mobile AND Multimedia Networks*, 2007.

[12] Dictionary.com LLC, 2009. http://dictionary.reference.com/.

[13] ANA Project Partners. Ip compartment. ANA Wiki, May 2009. https://www.ana-project.org/wiki/workpackages/wp1/task-1-6/integration/ip.

[14] Ion Stoica, Robert Morris, David Karger, M. Frans Kaashoek, and Hari Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*, San Diego, California, August 2001.

[15] Andrew Stuart Tanenbaum. *Computer Networks*. Pearson Education, fourth edition, 2003.

[16] Andrew Stuart Tanenbaum and Maarten Van Steen. *Distributed Systems - Principles and paradigms*. Pearson Education, second edition, 2007.

[17] Colby Walsworth, Emile Aben, kc claffy, and Dan Andersen. The caida anonymized 2009 internet traces - equinix-chicago.dira.20090331-055905.utc. http://www.caida.org/data/passive/passive_2009_dataset.xml.

[18] David A. Wheeler. *Linux Programmer's Manual*, July 2004. SLOCCOUNT(1).

APPENDIX

A Abbreviations

ANA	Autonomic Network Architecture
API	Application programming interface
CAIDA	The Cooperative Association for Internet Data Analysis
CBSE	Component-based software engineering
CPU	Central processing unit
DE	Data elements
DHT	Distributed hash table
DVD	Digital versatile disc
EL	External load
FB	Functional block
GB	Gigabyte
IC	Information channel
IDP	Information dispatch point
IMF	Integrated Monitoring Framework
IP	Internet Protocol
MB	Megabyte
MCIS	Multi Compartment Information Sharing
MHz	Megahertz
MinMex	Minimal Infrastructure for Maximal Extensibility
NAT	Network Address Translation
OS	Operating system
OSI Model	Open Systems Interconnection Basic Reference Model
PDA	Personal Digital Assistant
PDF	Portable Document Format
QPM	Queries per minute
RAM	Random-access memory
RIP	Routing Information Protocol
SL	System load
TTL	Time to live
WAN	Wide Area Network

Table 9: Abbreviations

B ANA code

The following subsections are dedicated to ANA programs used in this thesis. Each section represents one spesific brick.

B.1 node_monitor.c

```
1  /* ---------------------- Default brick header code ---------------------- */
2  /* brick_template.h contains the functions that initialize the bricks */
3  #include "brick_template.h"
4
5  /* Brick name */
6  char *mymodename="node_monitor";
7
8  /* Lisence */
9  #ifdef KERNEL
10 MODULE_LICENSE("Dual BSD/GPL");
11 MODULE_AUTHOR("ANA Project");
12 #endif
13 /* ---------------------- End default brick header code ---------------------- */
14
15 /* ---------------------- Brick specific code starts here ---------------------- */
16
17 // Origin for system information
18 #include <sys/sysinfo.h>
19
20 // ANA API container structs
21 static struct service_s service;
22 static struct context_s context;
23 static struct timespec timeout;
24
25 // System information struct
26 static struct sysinfo s_info;
27
28 // Percent usage of resource
29 static int cpu;
30 static int ram;
31
32 // IDPs
33 static anaLabel_t serviceIDP;
34
35 // Answer request with system information
36 void answer_request(struct anaL2_message *msg) {
37   anaPrint(ANA_DEBUG, "Answering request\n");
38
39   anaLabel_t requesterIDP = 0;
40
41   requesterIDP = msg->responseLabel;
42   if(requesterIDP == 0) {
43     anaPrint(ANA_ERR, "Can't find requesters IDP in incoming message\n");
44     return;
45   } else {
46     anaPrint(ANA_DEBUG, "Found requesters IDP '%x' in incoming message\n",
         requesterIDP);
47   }
48
49   char *reply_data = malloc(20);
```

```
50    sprintf(reply_data, "RAM:%d%% CPU:%d%%", ram, cpu);
51
52    if(anaL0_send(requesterIDP, reply_data, strlen(reply_data)) < 0) {
53      anaPrint(ANA_ERR, "Monitoring reply failed\n");
54      free(reply_data);
55      return;
56    }
57
58    anaPrint(ANA_NOTICE, "Monitoring data '%s' sent to '%x'\n", reply_data,
          requesterIDP);
59
60    free(reply_data);
61  }
62
63  // Timer based system inspection
64  void periodic_inspection() {
65    // s_info.loads[0] = 1 min average, [1] = 5 min average, [2] = 15 min average
66    sysinfo(&s_info);
67
68    int free = s_info.freeram / 1048576;
69    int total = s_info.totalram / 1048576;
70    int used = total - free;
71
72    ram = (used * 100) / total;
73    cpu = (int) (s_info.loads[0] / 1000);
74
75    anaPrint(ANA_DEBUG, "Inspection results - RAM:%d%% CPU:%d%%\n", ram, cpu);
76  }
77
78
79  // Shut down brick
80  void brick_exit() {
81    anaPrint(ANA_DEBUG, "Stopping node monitor brick\n");
82
83    // Unpublish the node monitor service
84    anaL2_unpublish(NODE_LABEL, serviceIDP, &context, &service, &timeout);
85  }
86
87  // Initialize brick
88  int brick_start() {
89    anaPrint(ANA_DEBUG, "Starting node monitor\n");
90
91    // Inspect system resources every 5 seconds
92    anatimer_init();
93    anatimer_add(5000, ANATIMER_PERIODIC, ANATIMER_ABSOLUTE, periodic_inspection,
          NULL, 0);
94
95    // Publish the node monitor service in the node compartment
96    timeout.tv_sec = 2;
97
98    context.value = "*";
99    context.valueLen = 2;
100
101   service.value = mymodename;
102   service.valueLen = strlen(service.value) + 1;
103
104   serviceIDP = anaL2_publish(NODE_LABEL, &context, &service, (AL2Callback_t) &
          answer_request, NOTHREAD, &timeout);
105   if(serviceIDP == 0) {
106     anaPrint(ANA_ERR, "Unable to publish node monitor service. Only logfile is
          available.\n");
107   } else {
```

```
108       anaPrint(ANA_DEBUG, "Published node monitor service in node compartment at
              IDP '%x'.\n", serviceIDP);
109     }
110
111     return 0;
112   }
```

Listing 5: Node monitor brick

B.2 decision_maker.cpp

```
1   /*------------------------- Default brick header code -------------------------*/
2   /* brick_template.h contains the functions that initialize the bricks */
3   extern "C" {
4     #include "brick_template.h"
5   }
6
7   /* Brick name */
8   char *mymodename="decision_maker";
9
10  /* Lisence */
11  #ifdef KERNEL
12  MODULE_LICENSE("Dual BSD/GPL");
13  MODULE_AUTHOR("ANA Project");
14  #endif
15  /*------------------------- End default brick header code ---------------------*/
16
17  /*------------------------- Brick specific code starts here-------------------*/
18  #include <signal.h>
19  #include <string>
20
21  using namespace std;
22
23  struct timespec timeout;
24
25  struct service_s monitor_service;
26  struct context_s monitor_context;
27
28  struct service_s my_service;
29  struct context_s my_context;
30
31  anaLabel_t monitorIDP = 0;
32  anaLabel_t myIDP = 0;
33
34  static unsigned int threshold_lower_cpu = 0;
35  static unsigned int threshold_upper_cpu = 0;
36  static unsigned int threshold_lower_ram = 0;
37  static unsigned int threshold_upper_ram = 0;
38
39  static int system_pid = 0;
40
41  /* Send signal to system allowing more resource consumption */
42  void increase_resources() {
43    printf("Signaling PID %d to increase resource consumption. Signal is '%d'.\n",
          system_pid, SIGUSR1);
44    anaPrint(ANA_NOTICE, "Signaling PID %d to increase resource consumption. Signal
          is '%d'.\n", system_pid, SIGUSR1);
45
46    int ret = kill(system_pid, SIGUSR1); // SIGUSR1 = MCIS' internal callback
          signal for increasing load
```

```
47    if(ret == -1) {
48      printf("Signaling PID %d failed!\n", system_pid);
49      anaPrint(ANA_ERR, "Signaling PID %d failed!\n", system_pid);
50    }
51  }
52
53  /* Send signal to system forcing less resource consumption */
54  void decrease_resources() {
55    printf("Signaling PID %d to decrease resource consumption. Signal is '%d'.\n",
          system_pid, 31);
56    anaPrint(ANA_NOTICE, "Signaling PID %d to decrease resource consumption. Signal
          is '%d'.\n", system_pid, 31);
57
58    int ret = kill(system_pid, 31); // 31 = MCIS' internal callback signal for
          decreasing load (31 instad of SIGUSR2 because Mercury threads are using
          SIGUSR2)
59    if(ret == -1) {
60      printf("Signaling PID %d failed!\n", system_pid);
61      anaPrint(ANA_ERR, "Signaling PID %d failed!\n", system_pid);
62    }
63  }
64
65  /* Parse the node monitor reply */
66  void handle_reply(struct anaL2_message *message) {
67    printf("Monitoring data received: '%s'\n", strndup((char *) message->data,
          message->dataLen));
68    anaPrint(ANA_NOTICE, "Monitoring data received:'%s'\n", strndup((char *)
          message->data, message->dataLen));
69
70    /* Parse response and set threshold values */
71    unsigned int actual_cpu = 0;
72    unsigned int actual_ram = 0;
73
74    /*RAM:79% CPU:11%*/
75    string data = strndup((char *) message->data, message->dataLen);
76    string::size_type ram_pos = data.find("RAM");
77    string::size_type cpu_pos = data.find("CPU");
78
79    actual_ram = (unsigned int) atoi(data.substr(ram_pos+4, cpu_pos-6).c_str());
80    actual_cpu = (unsigned int) atoi(data.substr(cpu_pos+4, data.size()-cpu_pos-5).
          c_str());
81
82    printf("Using %d%% RAM and %d%% CPU\n", actual_ram, actual_cpu);
83
84    /* Make decisions based on thresholds */
85    if(actual_cpu < threshold_lower_cpu) {
86      printf("Using %d%% CPU. Lower threshold is %d%%. Allowing system to use more
            resources.\n", actual_cpu, threshold_lower_cpu);
87      increase_resources();
88    } else if(actual_ram < threshold_lower_ram) {
89      printf("Using %d%% RAM. Lower threshold is %d%%. Allowing system to use more
            resources.\n", actual_ram, threshold_lower_ram);
90      increase_resources();
91    } else if(actual_cpu > threshold_upper_cpu) {
92      printf("Using %d%% CPU. Upper threshold is %d%%. Forcing system to use less
            resources.\n", actual_cpu, threshold_upper_cpu);
93      decrease_resources();
94    } else if(actual_ram > threshold_upper_ram) {
95      printf("Using %d%% RAM. Lower threshold is %d%%. Forcing system to use less
            resources.\n", actual_ram, threshold_upper_ram);
96      decrease_resources();
97    } else {
```

```
98        printf("Complying to thresholds. No signaling needed.\n");
99      }
100
101     printf("\n");
102   }
103
104   /* Timer based system inspection */
105   void periodic_request(void *) {
106     /* Ask for monitoring data */
107     anaPrint(ANA_DEBUG, "Requesting monitoring data from System monitor.\n");
108
109     xrpMsg_t request_msg = anaL1_allocateMessage();
110
111     int total_size = fillXRPCommand(request_msg, XRP_CMD_DATA);
112     if(total_size <= 0) {
113       anaPrint(ANA_ERR, "Could not insert data command in inspect message\n");
114       free(request_msg);
115       return;
116     } else {
117       anaPrint(ANA_DEBUG, "Inserted data command of size '%d' in inspect message\n"
              , total_size);
118     }
119
120     int argument_size = addXRPArg(request_msg, XRP_CLASS_MESSAGE, (char *) "Pliz",
          5);
121     if(argument_size <= 0) {
122       anaPrint(ANA_ERR, "Could not add begging to inspect message.\n");
123       free(request_msg);
124       return;
125     } else {
126       anaPrint(ANA_DEBUG, "Added begging to inspect message\n");
127     }
128     total_size += argument_size;
129
130     argument_size = addXRPArg(request_msg, XRP_CLASS_SRC_CXT, (char *) "*", 2);
131     if(argument_size <= 0) {
132       anaPrint(ANA_ERR, "Could not add my context to inspect message.\n");
133       free(request_msg);
134       return;
135     } else {
136       anaPrint(ANA_DEBUG, "Added my context to inspect message\n");
137     }
138     total_size += argument_size;
139
140
141     argument_size = addXRPArg(request_msg, XRP_CLASS_SRC_SRV, mymodename, strlen(
          mymodename));
142     if(argument_size <= 0) {
143       anaPrint(ANA_ERR, "Could not add my service to inspect message.\n");
144       free(request_msg);
145       return;
146     } else {
147       anaPrint(ANA_DEBUG, "Added my service '%s' to inspect message\n", mymodename)
              ;
148     }
149     total_size += argument_size;
150
151     argument_size = addXRPArg(request_msg, XRP_CLASS_LABEL, &myIDP, sizeof(
          anaLabel_t));
152     if(argument_size <= 0) {
153       anaPrint(ANA_ERR, "Could not add my IDP to inspect message.\n");
154       free(request_msg);
```

```
155       return;
156     } else {
157       anaPrint(ANA_DEBUG, "Added my IDP '%x' of size '%d' to inspect message\n",
              myIDP, argument_size);
158     }
159     total_size += argument_size;
160
161     int ret = anaL0_send(monitorIDP, request_msg, total_size);
162     if(ret < 0) {
163       anaPrint(ANA_ERR, "Sending request monitoring data message to System monitor
              IDP '%x' failed.\n", monitorIDP);
164     } else {
165       anaPrint(ANA_DEBUG, "Sending request monitoring data message of size '%d' to
              System monitor IDP '%x' succeeded.\n", total_size, monitorIDP);
166     }
167
168     free(request_msg);
169   }
170
171   /* Shut down brick */
172   void brick_exit() {
173     anaPrint(ANA_DEBUG, "Quitting decision maker brick.\n");
174
175     /* Unpublish our service */
176     anaL2_unpublish(NODE_LABEL, myIDP, &my_context, &my_service, &timeout);
177   }
178
179   /* Initialize brick */
180   int brick_start() {
181     /* Fetch variables from arguments */
182     system_pid = atoi(getAuxArg(0));
183     if(system_pid == 0) {
184       printf("No PID included.\n");
185       anaL0_stopANA(1);
186     }
187
188     threshold_lower_cpu = atoi(getAuxArg(1));
189     if(threshold_lower_cpu == 0) {
190       printf("No lower CPU threshold included.\n");
191       anaL0_stopANA(1);
192     }
193
194     threshold_upper_cpu = atoi(getAuxArg(2));
195     if(threshold_upper_cpu == 0) {
196       printf("No upper CPU threshold included.\n");
197       anaL0_stopANA(1);
198     }
199
200     threshold_lower_ram = atoi(getAuxArg(3));
201     if(threshold_lower_ram == 0) {
202       printf("No lower RAM threshold included.\n");
203       anaL0_stopANA(1);
204     }
205
206     threshold_upper_ram = atoi(getAuxArg(4));
207     if(threshold_upper_ram == 0) {
208       printf("No upper RAM threshold included.\n");
209       anaL0_stopANA(1);
210     }
211
212     printf("Signaling PID %d based on thresholds CPU:%d,%d and RAM:%d,%d\n",
          system_pid, threshold_lower_cpu, threshold_upper_cpu, threshold_lower_ram,
```

```
        threshold_upper_ram);
213     anaPrint(ANA_NOTICE, "Signaling PID %d based on thresholds CPU:%d,%d and RAM:%d
        ,%d\n", system_pid, threshold_lower_cpu, threshold_upper_cpu,
        threshold_lower_ram, threshold_upper_ram);
214
215     /* Common timeout values */
216     memset(&timeout, 0, sizeof(struct timespec));
217     timeout.tv_sec = 3;
218
219     /* Locate node monitor service in node compartment */
220     memset(&monitor_service, 0, sizeof(struct service_s));
221     memset(&monitor_context, 0, sizeof(struct context_s));
222
223     monitor_context.value = (char *) "*";
224     monitor_context.valueLen = 2;
225
226     monitor_service.value = (char *) "node_monitor";
227     monitor_service.valueLen = strlen((const char *) monitor_service.value) + 1;
228
229     monitorIDP = anaL2_resolve(NODE_LABEL, &monitor_context, &monitor_service, 'u',
          NULL, &timeout);
230     if(monitorIDP == 0) {
231       anaPrint(ANA_ERR, "Can't find node monitor.\n");
232       anaL0_stopANA(1);
233       return -1;
234     } else {
235       anaPrint(ANA_DEBUG, "Located node monitor.\n");
236     }
237
238     /* Make IDP available for monitoring feedback */
239     memset(&my_service, 0, sizeof(struct service_s));
240     memset(&my_context, 0, sizeof(struct context_s));
241
242     my_context.value = (char *) "*";
243     my_context.valueLen = 2;
244
245     my_service.value = (char *) mymodename;
246     my_service.valueLen = strlen((const char *) my_service.value) + 1;
247
248     myIDP = anaL2_publish(NODE_LABEL, &my_context, &my_service, (AL2Callback_t) &
          handle_reply, NOTHREAD, &timeout);
249     if(myIDP == 0) {
250       anaPrint(ANA_ERR, "Unable to make myself available to the System monitor.\n")
            ;
251       return -1;
252     } else {
253       anaPrint(ANA_DEBUG, "My IDP is '%x'.\n", myIDP);
254     }
255
256     /* Inspect system resources every 5 seconds */
257     anatimer_init();
258     anatimer_add(5000, ANATIMER_PERIODIC, ANATIMER_ABSOLUTE, periodic_request, NULL
          , 0);
259
260     return 1;
261   }
```

Listing 6: Decision maker brick

B.3 mcis_bootstrap.cpp

```
1  /*------------------------- Default brick header code -------------------------*/
2  /* brick_template.h contains the functions that initialize the bricks */
3  extern "C" {
4    #include "brick_template.h"
5  }
6  char *mymodename = (char *) "mcis_bootstrap";
7  #ifdef KERNEL
8  MODULE_LICENSE("Dual BSD/GPL");
9  MODULE_AUTHOR("ANA Project");
10 #endif
11 /*------------------------- End default brick header code ---------------------*/
12
13 /*------------------------- Brick specific code starts here-------------------*/
14
15 // Standard includes
16 #include <iostream>
17 #include <sstream>
18 #include <stdio.h>
19 #include <stdlib.h>
20 #include <unistd.h>
21
22 // Mercury specific includes
23 #include <mercury/BootstrapNode.h>
24 #include <Mercury.h>
25 #include "ana-env/ANANet.h"
26 #include <ana-env/ANAScheduler.h>
27 #include <util/OS.h>
28 #include <util/debug.h>
29 #include <util/TimeVal.h>
30 #include <mercury/options.h>
31 #include <mercury/Parameters.h>
32
33 static ANAScheduler *s_Scheduler;
34 static ANANet *s_Network;
35 static EndPoint s_BootstrapId;
36
37 // Hash list to store instances of nodes and their IDPs
38 template <char cp>
39 class char_sep : public unary_function<char, bool> {
40 public:
41   bool operator() (char c) const { return c == cp; }
42 };
43
44 static void DoWork(BootstrapNode *node) {
45   anaPrint(ANA_DEBUG, "Brick DoWork\n");
46
47   if(s_Network->StartListening(Parameters::TransportProto) != 0) {
48     anaPrint(ANA_ERR, "Unable to start the network '%d'\n", Parameters::
         TransportProto);
49     anaL0_stopANA(-1);
50   }
51
52   EndPoint *from = &noEp;
53   Message *msg = 0;
54
55   while(true) {
56     #ifndef ENABLE_ANANET_THREAD
57       ANANet::DoWork();
58     #endif
59
```

```
60      s_Scheduler->ProcessTill(s_Scheduler->TimeNow());
61
62      ConnStatusType status = s_Network->GetNextMessage(from, &msg);
63
64      switch(status) {
65        case CONN_NEWINCOMING:
66
67        case CONN_OK:
68          anaPrint(ANA_DEBUG, "Receiving message\n");
69          node->ReceiveMessage(from, msg);
70          break;
71
72        case CONN_CLOSED:
73          break;
74
75        case CONN_ERROR:
76          break;
77
78        case CONN_NOMSG:
79          break;
80
81        default:
82          anaPrint(ANA_ERR, "Unknown connection status.\n");
83          break;
84      }
85
86      OS::SleepMillis(100);
87    }
88  }
89
90  void brick_exit() {
91    anaPrint(ANA_DEBUG, "Brick EXIT\n");
92    anatimer_exit();
93  }
94
95  int brick_start() {
96    anaPrint(ANA_DEBUG, "Brick START\n");
97    int ret = 0;
98    int lenMsg, labelSize;
99    xrpMsg_t msg;
100   char *publ_name;
101
102   ret = anaL2_initDefault();
103
104   char **argVector = (char **) malloc(100 * sizeof(char *));
105   argVector[0] = "mcis_bootstrap";
106   int i = 1;
107   char *tmpArg = getAuxArg(i - 1);
108   while(tmpArg != NULL) {
109     argVector[i] = (char *) malloc(strlen(tmpArg) + 1);
110     strcpy(argVector[i], tmpArg);
111     tmpArg = getAuxArg(i++);
112   }
113
114   int argCount = i;
115
116   InitializeMercury(&argCount, argVector, g_BootstrapOptions, true);
117   DBG_INIT (NULL);
118
119   if (!g_BootstrapPreferences.schemaFile[0]) {
120     PrintUsage(g_BootstrapOptions);
121     return -1;
```

```
122     }
123
124     srand(42);
125
126     /* Start scheduler and network and create the node */
127     string hostName = string("mcis_bootstrap");
128     hostName = hostName + "_" + g_Preferences.hostname;
129     EndPoint s_BootstrapId((char *) hostName.c_str());
130
131     s_Scheduler = new ANAScheduler();
132
133     s_Network = new ANANet(s_Scheduler, s_BootstrapId);
134     if(s_Network == NULL) {
135       anaPrint(ANA_ERR, "Unable to create a new ANANet object\n");
136       return -1;
137     }
138
139     BootstrapNode *node = new BootstrapNode(s_Network, s_Scheduler, s_BootstrapId,
          g_BootstrapPreferences.schemaFile);
140
141     s_Scheduler->SetNode(node);
142
143     /* Start the node and go into work-lopp */
144     node->Start();
145     DoWork(node);
146
147     return 0;
148 }
```

Listing 7: MCIS Bootstrap

B.4 mcis_peer.cpp

```
1  /*------------------------- Default brick header code -------------------------*/
2  /* brick_template.h contains the functions that initialize the bricks */
3  extern "C" {
4    #include "brick_template.h"
5  }
6  char *mymodename = (char *) "mcis_peer";
7  #ifdef KERNEL
8  MODULE_LICENSE("Dual BSD/GPL");
9  MODULE_AUTHOR("ANA Project");
10 #endif
11 /*------------------------- End default brick header code ---------------------*/
12
13 /*------------------------- Brick specific code starts here-------------------*/
14 struct service_s service_ip;
15 struct service_s service;
16 struct context_s context;
17 struct timespec timeout;
18
19 //standard includes
20 #include <iostream>
21 #include <sstream>
22 #include <unistd.h>
23 #include <signal.h>
24
25 //mercury specific includes
26 #include <Mercury.h>
27 #include <ana-env/ANANet.h>
```

```
28 #include <mercury/ObjectLogs.h>
29 #include <mercury/RoutingLogs.h>
30 #include <ana-env/ANAMercuryNode.h>
31 #include <mercury/Hub.h>
32 #include <mercury/HubManager.h>
33 #include <mercury/Sampling.h>
34 #include <util/OS.h>
35 #include <mercury/options.h>
36 #include <mercury/Event.h>
37 #include <mercury/Application.h>
38 #include <mercury/PubsubStore.h>
39
40
41 char *pubNameExt;
42 static anaLabel_t ipIDP;
43 anaLabel_t recvDataIDP;
44 anaLabel_t coreInput;
45 ANAMercuryNode *m_Router;
46
47 map<string, anaLabel_t> mercIDPs;
48 map<string, anaLabel_t>::iterator mercIDPsIter;
49
50 //map<string, ANAMercuryNode *> mercNodes;
51 //map<string, ANAMercuryNode *>::iterator mercNodesIter;
52
53 map<anaLabel_t, ANAMercuryNode *> mercNodes;
54 map<anaLabel_t, ANAMercuryNode *>::iterator mercNodesIter;
55
56 map<anaLabel_t, char *> mercSchemas;
57 map<anaLabel_t, char *>::iterator mercSchemasIter;
58
59 map<string, anaLabel_t> clients; // Clients using MCIS. String is concationation
       of description and ANAIP
60 map<string, anaLabel_t>::iterator clientsIter;
61
62 struct LookupArgs {
63   anaLabel_t input;
64   char *data;
65   int len;
66   uint32  size;
67 };
68
69 template <char cp>
70 class char_sep : public unary_function<char, bool> {
71  public:
72   bool operator() (char c) const { return c == cp; }
73 };
74
75 anaLabel_t find_client(char *client_ip, char *client_service);
76 void recvFromHL(struct anaL2_message *msg);
77 int startNewMerc(char *dataType, char *joinLocation, char *schema/*, anaLabel_t
       mercInput*/);
78
79 //void handlePublishReply(char * data, int len, anaLabel_t input, void *aux);
80 void handleUnpublishReply(char *data, int len, anaLabel_t input, void *aux);
81
82 void handlePublish(anaLabel_t input, void *msg, int len);
83 void handleUnpublish(void *msg, int len);
84 void handleResolve(anaLabel_t input, void *msg, int len);
85 void handleLookup(void *argus);
86
87 void handlePublishCore(void *msg, int len);
```

```
88  void handleUnpublishCore(void *msg, int len);
89  void handleResolveCore(void *data, int len);
90  void handleLookupCore(void *argus);
91
92  #define LOOKUP_TO 10
93  #define RESOLVE_TO 5
94
95  //-MISC FUNCTIONS-------------------------------------------------------------
96  /* Signal handlers for resource adaptation */
97  void sigusr1_handler(int signal) {
98    printf("Received signal '%d'. Increasing resource consumption!\n", signal);
99    anaPrint(ANA_NOTICE, "Received signal '%d'. Increasing resource consumption!\n"
          , signal);
100
101   /* Tell all the data compartments to adjust */
102   for(mercNodesIter = mercNodes.begin(); mercNodesIter != mercNodes.end();
          mercNodesIter++) {
103     anaPrint(ANA_DEBUG, "Data compartment IDP is '%x'\n", mercNodesIter->first);
104     mercNodesIter->second->UseResources();
105   }
106 }
107
108 void sigusr2_handler(int signal) {
109   printf("Received signal '%d'. Decreasing resource consumption!\n", signal);
110   anaPrint(ANA_NOTICE, "Received signal '%d'. Decreasing resource consumption!\n"
          , signal);
111   /* Tell all the data compartments to adjust */
112   for(mercNodesIter = mercNodes.begin(); mercNodesIter != mercNodes.end();
          mercNodesIter++) {
113     anaPrint(ANA_DEBUG, "Data compartment IDP is '%x'\n", mercNodesIter->first);
114     mercNodesIter->second->ReleaseResources();
115   }
116 }
117
118 /* Resolves the client with given IP and description.
119  * Either in local cache or in IP compartment if previosly unknown.
120  */
121 anaLabel_t find_client(char *client_ip, char *client_service) {
122   anaLabel_t clientIDP = 0;
123
124   string client_id = (string) client_ip + (string) client_service;
125
126   clientIDP = clients[client_id];
127   if(clientIDP == 0) {
128     anaPrint(ANA_DEBUG, "Did not find client in cache. Need to resolve '%s' at IP
          '%s'\n", client_service, client_ip);
129     struct service_s service;
130     struct context_s context;
131     struct timespec timeout;
132
133     memset (&service, 0, sizeof(struct service_s));
134     memset (&context, 0, sizeof(struct context_s));
135     memset (&timeout, 0, sizeof(struct timespec));
136
137     service.value = (void *) client_service;
138     service.valueLen = strlen((const char*) service.value) + 1;
139
140     context.value = (void *) client_ip;
141     context.valueLen = strlen((const char*) context.value) + 1;
142
143     timeout.tv_sec = 5;
144
```

```
145       clientIDP = anaL2_resolve(ipIDP, &context, &service, 'u', NULL, &timeout);
146       if(clientIDP == 0) {
147         anaPrint(ANA_ERR, "ERROR: Discovering client '%s' at '%s' in IP compartment
                failed\n", client_service, client_ip);
148       } else {
149         clients[client_id] = clientIDP;
150       }
151     } else {
152       anaPrint(ANA_DEBUG, "Found client in cache at IDP '%x'\n", clientIDP);
153     }
154
155     return clientIDP;
156
157   }
158
159   /*
160    * Wrapper
161    * Puts L1 data in anaL2_message struct and calls real callback function
162   */
163   /*
164   void recvFromHL_wrapper(char *data, int len, anaLabel_t input, AL2Callback_t aux)
          {
165     anaPrint(ANA_DEBUG, "Wrapping L1 data in L2 message\n");
166
167     struct anaL2_message msg;
168     anaLabel_t *tmpLabel = NULL;
169     struct context_s senderContext;
170     struct service_s senderService;
171
172     if (data == NULL) {
173       anaPrint(ANA_ERR, "No data\n");
174       return;
175     }
176
177     memset(&msg, 0, sizeof(struct anaL2_message));
178
179     msg.idp = input;
180
181     if (equalXRPCommands((char *) data, XRP_CMD_DATA)){
182       memset(&senderContext, 0, sizeof(struct context_s));
183       memset(&senderService, 0, sizeof(struct service_s));
184
185       msg.dataLen = anaL1_decResponse((xrpMsg_t)data, XRP_CLASS_MESSAGE, 0, &(msg.
              data));
186
187       senderContext.valueLen = anaL1_decResponse((xrpMsg_t)data, XRP_CLASS_SRC_CXT,
              0, &(senderContext.value));
188       senderService.valueLen = anaL1_decResponse((xrpMsg_t)data, XRP_CLASS_SRC_SRV,
              0, &(senderService.value));
189
190       msg.senderContext = &senderContext;
191       msg.senderService = &senderService;
192
193       anaL1_decResponse((xrpMsg_t)data, XRP_CLASS_LABEL, 0, (void**) &tmpLabel);
194
195       if(tmpLabel) {
196         msg.responseLabel = *tmpLabel;
197       }
198     } else {
199       msg.dataLen = len;
200       msg.data = data;
201     }
```

```
202
203     // Call real function with encapsulated message
204     recvFromHL(&msg);
205   }
206   */
207
208   /*
209    * Callback for incoming messages.
210    * Forwards compartment API messages to handler functions corresponding to the
         XRP_CMD specified
211   */
212   void recvFromHL(struct anaL2_message *msg) {
213     xrpCmd_t command = (char *) msg->data;
214     anaLabel_t input = msg->idp;
215     char *data = (char *) msg->data;
216     int len = msg->dataLen;
217
218     if(equalXRPCommands(command, XRP_CMD_PUBLISH)) {
219       // PUBLISH
220       anaPrint(ANA_DEBUG, "Received publish request\n");
221
222       if(input == coreInput) {
223         // Publish inside metadata compartment (start new data compartment)
224         handlePublishCore(data, len);
225       } else {
226         // Publish inside data compartment (store data element)
227         handlePublish(input, data, len);
228       }
229     } else if(equalXRPCommands(command, XRP_CMD_UNPUBLISH)){
230       // UNPUBLISH
231       anaPrint(ANA_DEBUG, "Received unpublish request\n");
232
233       if(input == coreInput) {
234         // Unpublish from metadata compartment (Shut down data compartment)
235         handleUnpublishCore(data, len);
236       } else {
237         // Unpublish from data compartment (delete data element)
238         handleUnpublish(data, len);
239       }
240     } else if(equalXRPCommands(command, XRP_CMD_RESOLVE)) {
241       // RESOLVE
242       anaPrint(ANA_DEBUG, "Received resolve request\n");
243
244       if(input == coreInput) {
245         // Resolve in metadata compartment (find data compartment)
246         handleResolveCore(data, len);
247       } else {
248         // Resolve in data compartment (find IDPs to nodes with data element)
249         handleResolve(input, data, len);
250       }
251     } else if(equalXRPCommands(command, XRP_CMD_LOOKUP)) {
252       // LOOKUP
253       anaPrint(ANA_DEBUG, "Received lookup request\n");
254
255       pthread_t lookupThread;
256       struct LookupArgs *argus = (struct LookupArgs *) malloc(sizeof(struct
            LookupArgs));
257       argus->input = input;
258       argus->data = data;
259       argus->len = len;
260
261       if(input == coreInput) {
```

```
262         // Lookup in metadata compartment
263         pthread_create(&lookupThread, NULL, (void*(*)(void*)) handleLookupCore, (
                void *) argus);
264       } else {
265         // Lookup in data compartment (find data elements / query)
266         pthread_create(&lookupThread, NULL, (void*(*)(void*)) handleLookup, (void
                *) argus);
267       }
268     } else {
269       anaPrint(ANA_ERR, "Received unknown message to IDP '%x' Message='%s'\n",
              input, command);
270     }
271   }
272
273   /*
274    * Create and initiate a new MercuryNode for the given data type
275    * Stores the IDP and the pointer to the instance into a map
276    */
277   int startNewMerc(char *dataType, char *joinLocation, char *schema/*, anaLabel_t
          mercInput*/) {
278     anaPrint(ANA_NOTICE, "Starting data compartment node with schema='%s'\n",
            schema);
279     void *msg;
280     int number_of_hubs = 0;
281     anaLabel_t mercInput = 0;
282
283     // Start the Mercury node for the published datatype
284     string data_compartment_name = string("dc_") + string(dataType) + string("_") +
             string(pubNameExt); // This is for communication between peers, not for
            clients/users
285     anaPrint(ANA_DEBUG, "Data compartment name '%s'\n", data_compartment_name.c_str
            ());
286     ANAMercuryNode *router = ANAMercuryNode::GetInstance(data_compartment_name.
            c_str());
287     router->SetID(data_compartment_name.c_str());
288
289     // Check if this is the first to join the hubs ie. set the join locations to
            ourselves
290     if(joinLocation == NULL) {
291       anaPrint(ANA_DEBUG, "Lone node. Join ourself.\n");
292       joinLocation = (char *) malloc(1024 * sizeof(char));
293       vector<string> js;
294       tokenizer<comma_sep>::tokenize(js, schema);
295
296       // Go through all the attributes in the schema
297       for (uint32 i = 0; i < js.size(); i++) {
298         number_of_hubs++;
299         vector<string> inf;
300         //identify lower and upper bounds
301         tokenizer<char_sep<':'> >::tokenize(inf, js[i]);
302         if (i > 0) {
303           sprintf(joinLocation, "%s,%s:%s", joinLocation, inf[0].c_str(), router->
                GetHubManager()->GetAddress().ToString());
304         } else {
305           sprintf(joinLocation, "%s:%s", inf[0].c_str(), router->GetHubManager()->
                GetAddress().ToString());
306         }
307       }
308     } else {
309       anaPrint(ANA_DEBUG, "Not a loner.\n");
310       // Need to set number of hubs even if joinLocation is set.
311       string js = string(joinLocation);
```

```
312       number_of_hubs = std::count(js.begin(), js.end(), ',');
313       // x,y,z = 3, not 2
314       number_of_hubs++;
315     }
316
317     anaPrint(ANA_DEBUG, "Join location is '%s'\n", joinLocation);
318
319     strcpy(g_Preferences.join_locations, joinLocation);
320     strcpy(g_Preferences.schema_string, schema);
321     g_Preferences.bootstrap[0] = '\0';
322
323     // Start the router (node) with the metadata (datatype) store
324     anaPrint(ANA_NOTICE, "Number of hubs is '%d'\n", number_of_hubs);
325     DummyApp *m_app = new DummyApp(number_of_hubs);
326     router->RegisterApplication(m_app);
327
328     // Start the thread for the router and consequently for the "network" (ANANet)
329     router->FireUp();
330
331     // Send a ping to the join location of the first attribute hub
332     // If its us (new data cmpt) we just send it to ourself and reply to ourself
          too.
333     vector<string> inf, js;
334     tokenizer<comma_sep>::tokenize(js, joinLocation);
335     tokenizer<char_sep<':'> >::tokenize(inf, js[0]);
336
337     anaPrint(ANA_DEBUG, "Sending PING to EndPoint '%s'\n", inf[1].c_str());
338     while(router->SendPing(EndPoint((char *) inf[1].c_str())) == -1) {
339       #ifndef ENABLE_ANANET_THREAD
340         ANANet::DoWork (100);
341       #else
342         OS::SleepMillis(100);
343       #endif
344
345       #ifndef HAVE_THREADS
346         router->DoWork (200);
347       #else
348         OS::SleepMillis(200);
349       #endif
350     }
351
352     anaPrint(ANA_DEBUG, "Sent PING\n");
353
354     while(!router->AllJoined()) {
355       #ifndef ENABLE_ANANET_THREAD
356         ANANet::DoWork (100);
357       #else
358         OS::SleepMillis(100);
359       #endif
360
361       #ifndef HAVE_THREADS
362         router->DoWork (200);
363       #else
364         OS::SleepMillis(200);
365       #endif
366
367       OS::SleepMillis(300);
368     }
369
370     anaPrint(ANA_NOTICE, "JOINED!\n");
371
```

```
372     // Register callback function for clients to find data compartment in IP
            compartment (ie. not for internal communication between peers)
373     // XXX Old mercInput = anaL0_registerCallback((anaCallback_t)
            recvFromHL_wrapper, NULL, NULL, NULL, 0, IDP_PERM, THREAD);
374     struct service_s service;
375     struct context_s context;
376     struct timespec timeout;
377
378     memset(&service, 0, sizeof(struct service_s));
379     memset(&context, 0, sizeof(struct context_s));
380     memset(&timeout, 0, sizeof(struct timespec));
381
382     timeout.tv_sec = 3;
383
384     context.value = (void *) "*";
385     context.valueLen = 2;
386
387     string data_compartment_name_external = string("compartment+
            mcis_data_compartment+") + string(dataType) + string("+") + string(
            pubNameExt);
388     service.value = (void *) data_compartment_name_external.c_str();
389     service.valueLen = strlen((const char*) service.value) + 1;
390
391     mercInput = anaL2_publish(ipIDP, &context, &service, (AL2Callback_t) &
            recvFromHL, THREAD, &timeout);
392     if(mercInput == 0) {
393       anaPrint(ANA_ERR, "ERROR: Unable to publish data compartment '%s' in ip
              compartment\n", data_compartment_name_external.c_str());
394       return -1;
395     }
396
397     // Add the receive IDP of the new Mercury instance to the map
398     mercIDPs[dataType] = mercInput;
399
400     // Add the mercurynode itself to the map
401     mercNodes[mercInput] = router;
402
403     // Add the schema to the map
404     mercSchemas[mercInput] = schema;
405
406     anaPrint(ANA_NOTICE, "New data compartment available at local IDP='%x'\n",
            mercInput);
407
408     return 0;
409   }
410
411   /* Checks whether the brick was successfully published */
412   /*
413   void handlePublishReply(char *data, int len, anaLabel_t input, void *aux) {
414     char *name = NULL;
415     int nameLen = anaL1_decResponse(data, XRP_CLASS_SRC_SRV, 0, (void**) &name);
416
417     if(anaL1_isError(data)) {
418       if(name != NULL) {
419         anaPrint(ANA_ERR, "Publish of entry %s failed\n", name);
420       } else {
421         anaPrint(ANA_ERR, "Publish error\n");
422         return;
423       }
424     } else {
425       if(name != NULL) {
426         anaPrint(ANA_DEBUG, "Publish of entry %s succeeded\n", name);
```

```
427       } else {
428         anaPrint(ANA_DEBUG, "Publish success\n");
429       }
430     }
431
432     INFO << "MCIS " << pubNameExt << " is ready..." << endl;
433   }
434   */
435
436   /* Checks whether the brick was successfully unpublished */
437   void handleUnpublishReply(char *data, int len, anaLabel_t input, void *aux) {
438     if(anaL1_isError(data)) {
439       anaPrint(ANA_ERR, "Unpublish Error\n");
440     } else {
441       anaPrint(ANA_DEBUG, "Unpublish Success\n");
442     }
443   }
444
445   //--DATA COMPARTMENT HANDLER FUNCTIONS-----------------------------------------------
446   /* Store data element */
447   void handlePublish(anaLabel_t input, void *msg, int len) {
448     anaPrint(ANA_DEBUG, "PUBLISH - In data compartment node '%s' with IDP '%x'.\n",
          pubNameExt, input);
449
450     anaLabel_t replyTo = 0;
451     char *dataRecord = NULL;
452     int dummy = 0;
453
454     ANAMercuryNode *router = mercNodes[input];
455     if(router == 0) {
456       anaPrint(ANA_ERR, "ERROR: No such data router.\n");
457       return;
458     }
459
460   /*
461     int labelLen = 0;
462     anaLabel_t *labelP;
463     labelLen = anaL1_decResponse((char *) msg, XRP_CLASS_LABEL , 1, (void **) &
          labelP);
464     replyTo = *labelP;
465   */
466
467     dataRecord = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_SRC_SRV, &dummy, 0);
468     if(dataRecord == NULL) {
469       anaPrint(ANA_ERR, "ERROR: Unable to identify data record.\n");
470       return;
471     }
472
473     char *client_ip = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_MESSAGE, &dummy,
          0);
474     anaPrint(ANA_DEBUG, "Client IP '%s'\n", client_ip);
475     char *client_service = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_SRC_SRV, &
          dummy, 1);
476     anaPrint(ANA_DEBUG, "Client service '%s'\n", client_service);
477
478     replyTo = find_client(client_ip, client_service);
479     anaPrint(ANA_DEBUG, "Reply to IDP '%x'\n", replyTo);
480
481     // Create the event and send it out
482     PointEvent *ev = new PointEvent();
483     vector<string> js;
484     anaPrint(ANA_DEBUG, "Creating pointevent from '%s'\n", dataRecord);
```

```
485     tokenizer<comma_sep>::tokenize(js, dataRecord);
486
487     // Go through all the attributes in the schema
488     for (uint32 i = 0; i < js.size(); i++) {
489       vector<string> inf;
490       // Identify lower and upper bounds
491       tokenizer<char_sep<':'> >::tokenize(inf, js[i]);
492       //unsigned int minVal, maxVal; // XXX Hans: This needs to be fixed for large
              values and strings!
493       string tmpmin, tmpmax;
494       stringstream ss((string) inf[0]);
495       //ss >> minVal;
496       ss >> tmpmin;
497       MercuryID minVal((const char *) tmpmin.c_str()); // XXX Hans: Possible
              solution
498       ss.clear();
499       if(inf.size() > 1) {
500         ss.str((string)inf[1]);
501         //ss >> maxVal;
502         ss >> tmpmax;
503         MercuryID maxVal((const char *) tmpmax.c_str()); // XXX Hans: Possible
              solution
504         anaPrint(ANA_DEBUG, "Constraint min='%s' max='%s'\n", tmpmin.c_str(),
              tmpmax.c_str());
505         Constraint c(i, minVal, maxVal);
506         ev->AddConstraint(c);
507       } else {
508         anaPrint(ANA_DEBUG, "Constraint min='%s'\n", tmpmin.c_str());
509         Constraint c(i, minVal, minVal);
510         ev->AddConstraint(c);
511       }
512
513       // Make the data item permanent
514       ev->SetLifeTime(-1);
515     }
516
517     router->SendEvent(ev);
518     delete ev;
519
520     // Send the same message as success reply
521     anaPrint(ANA_DEBUG, "Sending publish scuccess to IDP '%x'.\n", replyTo);
522
523     // Send back the same received message
524     if(replyTo != 0) {
525       anaL0_send(replyTo, msg, len);
526     }
527
528     return;
529   }
530
531   /* Delete data element */
532   void handleUnpublish(void *msg, int len) {
533     anaPrint(ANA_DEBUG, "UNPUBLISH - In data compartment node '%s'.\n", pubNameExt)
          ;
534
535     anaLabel_t replyTo;
536     //anaLabel_t *labelP;
537     char *dataRecord = NULL;
538     int dummy = 0;
539
540   /*
541     int labelLen;
```

```
542     labelLen = anaL1_decResponse((char *) msg, XRP_CLASS_LABEL , 1, (void **) &
            labelP);
543     replyTo = *labelP;
544   */
545
546     char *client_ip = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_MESSAGE, &dummy,
            0);
547     anaPrint(ANA_DEBUG, "Client IP '%s'\n", client_ip);
548     char *client_service = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_SRC_SRV, &
            dummy, 1);
549     anaPrint(ANA_DEBUG, "Client service '%s'\n", client_service);
550
551     replyTo = find_client(client_ip, client_service);
552     anaPrint(ANA_DEBUG, "Reply to IDP '%x'\n", replyTo);
553
554     dataRecord = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_SRC_SRV, &dummy, 0);
555     if(dataRecord == NULL) {
556       anaPrint(ANA_ERR, "Unable to identify data record.\n");
557       return;
558     }
559
560     // Create the event and send it out
561     PointEvent *ev = new PointEvent();
562     vector<string> js;
563     anaPrint(ANA_DEBUG, "Creating pointevent from '%s'\n", dataRecord);
564     tokenizer<comma_sep>::tokenize(js, dataRecord);
565
566     // Go through all the attributes in the schema
567     for (uint32 i = 0; i < js.size(); i++) {
568       vector<string> inf;
569       // Identify lower and upper bounds
570       tokenizer<char_sep<':'> >::tokenize(inf, js[i]);
571       //unsigned int minVal, maxVal; // XXX Hans: This needs to be fixed for large
              values and strings!
572       string tmpmin, tmpmax; // XXX Hans: Possible solution
573       stringstream ss((string)inf[0]);
574       //ss >> minVal;
575       ss >> tmpmin;
576       MercuryID minVal((const char *) tmpmin.c_str()); // XXX Hans: Possible
              solution
577       ss.clear();
578       if(inf.size() > 1) {
579         ss.str((string)inf[1]);
580         //ss >> maxVal;
581         ss >> tmpmax;
582         MercuryID maxVal((const char *) tmpmax.c_str()); // XXX Hans: Possible
                solution
583         anaPrint(ANA_DEBUG, "Constraint min='%s' max='%s'\n", tmpmin.c_str(),
                tmpmax.c_str());
584         Constraint c(i, minVal, maxVal);
585         ev->AddConstraint(c);
586       } else {
587         anaPrint(ANA_DEBUG, "Constraint min='%s'\n", tmpmin.c_str());
588         Constraint c(i, minVal, minVal);
589         ev->AddConstraint(c);
590       }
591
592       // Make the data item permanent
593       ev->SetLifeTime(-1);
594     }
595
596     // Send remove event to all known routers
```

```
597     anaPrint(ANA_DEBUG, "Sending delete command to '%d' peers.\n", mercNodes.size()
            );
598     for(mercNodesIter = mercNodes.begin(); mercNodesIter != mercNodes.end();
            mercNodesIter++ ) {
599       ANAMercuryNode *router = mercNodesIter->second;
600
601       if (router == NULL ) {
602         continue;
603       }
604
605       router->SendUnPublishEvent(ev);
606     }
607
608     delete ev;
609
610     // Send the same message as success reply
611     anaPrint(ANA_DEBUG, "Sending unpublish scuccess to IDP '%x'.\n", replyTo);
612
613     // Send back the same received message
614     if(replyTo != 0) {
615       anaL0_send(replyTo, msg, len);
616     }
617
618     return;
619   }
620
621   /* Find IDPs to nodes which have the queried data */
622   void handleResolve(anaLabel_t input, void *msg, int len) {
623     anaPrint(ANA_DEBUG, "RESOLVE - In data compartment\n");
624     //one option is to return a set of IDPs that are end points of ICs to nodes
            which have the data we queried
625     //but think about this later...
626     return;
627   }
628
629   /* Find data elements */
630   void handleLookup(void *argus) {
631     anaPrint(ANA_DEBUG, "LOOKUP - In data compartment\n");
632
633     struct LookupArgs *lookup_args = (struct LookupArgs *) argus;
634     xrpMsg_t msg = lookup_args->data;
635     anaLabel_t input = lookup_args->input;
636     int len = lookup_args->len;
637
638     anaLabel_t replyTo = 0;
639   /* anaLabel_t *labelP; */
640     void *query = NULL;
641     int dummy = 0;
642   /*
643     int labelSize = anaL1_decResponse((char *) msg, XRP_CLASS_LABEL , 0, (void **)
            &labelP);
644     replyTo = *labelP;
645     anaPrint(ANA_NOTICE, "Reply to IDP '%x'.\n", replyTo);
646   */
647
648     char *client_ip = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_MESSAGE, &dummy,
            0);
649     anaPrint(ANA_DEBUG, "Client IP '%s'\n", client_ip);
650     char *client_service = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_SRC_SRV, &
            dummy, 0);
651     anaPrint(ANA_DEBUG, "Client service '%s'\n", client_service);
652
```

```
653     replyTo = find_client(client_ip, client_service);
654     anaPrint(ANA_DEBUG, "Reply to IDP '%x'\n", replyTo);
655
656     query = (char *) getXRPArg(msg, XRP_CLASS_DST_SRV, &dummy, 0);
657     if(query == NULL) {
658       anaPrint(ANA_ERR, "Unable to identify query.\n");
659       return;
660     } else {
661       anaPrint(ANA_DEBUG, "Making query='%s'.\n", (char *) query);
662     }
663
664     char *tmpnonce = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_MESSAGE, &dummy,
          1);
665     string nonce = tmpnonce;
666     anaPrint(ANA_DEBUG, "NONCE '%s'\n", nonce.c_str());
667
668     ANAMercuryNode *router = mercNodes[input];
669     if (router == 0) {
670       anaPrint(ANA_ERR, "No such data router.\n");
671       return;
672     }
673
674     // Process the comma separated constraints and semicolon separated bounds and
          create the constraint objects
675     // Query must be in the form: "attr_nb:min:max,attr_nb:min:max,..."
676     Query *q = new Query();
677     vector<string> con;
678     tokenizer<comma_sep>::tokenize(con, (char *) query);
679     Constraint consts[con.size()];
680
681     for(int i = 0, n = con.size(); i < n; i++) {
682       vector<string> inf;
683       tokenizer<char_sep<':'> >::tokenize(inf, con[i]);
684       //unsigned int attrNb, minVal, maxVal; // Hans XXX Anoher instance where
            large values might be a problem
685       int attrNb;
686       string tmpmin, tmpmax; // Hans XXX Possible solution
687       stringstream ss((string)inf[0]);
688       ss >> attrNb;
689       ss.clear();
690       ss.str((string)inf[1]);
691       //ss >> minVal;
692       ss >> tmpmin;
693       MercuryID minVal((const char *) tmpmin.c_str()); // XXX Hans: Possible
            solution
694       ss.clear();
695       ss.str((string)inf[2]);
696       //ss >> maxVal;
697       ss >> tmpmax;
698       MercuryID maxVal((const char *) tmpmax.c_str()); // XXX Hans: Possible
            solution
699       consts[i] = Constraint(attrNb, minVal, maxVal);
700       anaPrint(ANA_DEBUG, "Attribute='%d' MIN='%s' MAX='%s'.\n", attrNb, tmpmin.
            c_str(), tmpmax.c_str());
701       q->AddConstraint(consts[i]);
702     }
703
704     stringstream tmpquery;
705     tmpquery << q;
706     anaPrint(ANA_DEBUG, "%s\n", tmpquery.str().c_str());
707
708     // Send the query into the data compartment
```

```
709     router->RegisterQuery(q);
710     delete q, consts;
711
712     // Make a reply message
713     xrpMsg_t reply_msg = anaL1_allocateMessage();
714     // REDUNDANT memset(reply_msg, 0, defaultXrpSpecs.maxMsgSize);
715     // REDUNDANT allocateXRPMsg(&reply_msg);
716     int totalSize, arglen;
717     totalSize = fillXRPCommand(reply_msg, XRP_CMD_DATA);
718
719     // Check if we receive query results
720     Event *e = NULL;
721     TimeVal v;
722     gettimeofday(&v, NULL);
723     uint64 nowtime = ((uint64) v.tv_sec * USEC_IN_SEC) + v.tv_usec;
724     uint64 stopTime = nowtime + (unsigned long long) LOOKUP_TO * 1000000;
725
726     while(nowtime < stopTime) {
727       e = router->ReadEvent();
728       if(e != NULL) {
729         anaPrint(ANA_DEBUG, "Result received!\n");
730         // Send this result tuple back to the query requester
731         char *qResult = (char *) malloc(1024 * sizeof(char));
732         memset(qResult, '\0', 1024);
733
734         // Creating data element from all the constraints
735         anaPrint(ANA_DEBUG, "Result has %d attributes.\n", e->GetNumConstraints());
736
737         stringstream tmpval;
738         anaPrint(ANA_DEBUG, "NONCE=='%s'\n", nonce.c_str());
739         tmpval << "NONCE " << nonce << " - " << e->GetConstraint(0)->GetMin();
740         for(int i = 1; i < e->GetNumConstraints(); i++) {
741           tmpval << "," << e->GetConstraint(i)->GetMin();
742         }
743
744         sprintf(qResult, "%s", tmpval.str().c_str());
745
746         anaPrint(ANA_DEBUG, "Size='%d' Result='%s'\n", strlen(qResult), qResult);
747
748         arglen = addXRPArg(reply_msg, XRP_CLASS_LABEL, &replyTo, sizeof(anaLabel_t)
                 ); // Hans XXX Dunno which label this should be, but the other end
                 expects a label here...
749         if(arglen < 0) {
750           anaPrint(ANA_ERR, "Unable to label to message\n");
751           free(reply_msg);
752           return;
753         }
754         totalSize += arglen;
755
756         arglen = addXRPArg(reply_msg, XRP_CLASS_LKP_DESCR, (char *) qResult, strlen
                 (qResult) + 1);
757         if(arglen < 0) {
758           anaPrint(ANA_ERR, "Unable to add result to message\n");
759           free(reply_msg);
760           return;
761         }
762         totalSize += arglen;
763
764         free(qResult);
765       }
766
767       OS::SleepMillis(100);
```

```
768       gettimeofday(&v, NULL);
769       nowtime = ((uint64) v.tv_sec * USEC_IN_SEC) + v.tv_usec;
770     }
771
772     // Send the reply message
773     anaPrint(ANA_DEBUG, "Sending message of size '%d' to IDP '%x'.\n", totalSize,
            replyTo);
774     anaL0_send(replyTo, reply_msg, totalSize);
775     free(reply_msg);
776
777
778     pthread_exit(NULL);
779
780     return;
781   }
782
783   //--METADATA COMPARTMENT HANDLER FUNCTIONS-----------------------------------------
784   /* Start new data compartment */
785   void handlePublishCore(void *msg, int len) {
786     anaPrint(ANA_DEBUG, "PUBLISH - In metadata compartment\n");
787
788     anaLabel_t replyTo = 0;
789     //anaLabel_t newIDP = 0;
790     anaLabel_t *labelP = 0;
791     xrpMsg_t replyMsg;
792     int descLen = 0;
793     int dummy = 0;
794     char *dataTypeSchema = NULL;
795     char *dataType = NULL;
796     char *schema = NULL;
797     char *joinLocations = NULL;
798
799     dataTypeSchema = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_SRC_SRV, &descLen
            , 0);
800     if(dataTypeSchema == NULL) {
801       anaPrint(ANA_ERR, "Unable to identify published data type and schema\n");
802       return;
803     } else {
804       anaPrint(ANA_NOTICE, "Schema='%s'\n", dataTypeSchema);
805     }
806
807     char *client_ip = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_MESSAGE, &dummy,
            0);
808     anaPrint(ANA_DEBUG, "Client IP '%s'\n", client_ip);
809     char *client_service = (char *) getXRPArg((xrpMsg_t) msg, XRP_CLASS_SRC_SRV, &
            dummy, 1);
810     anaPrint(ANA_DEBUG, "Client service '%s'\n", client_service);
811
812     replyTo = find_client(client_ip, client_service);
813     anaPrint(ANA_DEBUG, "Reply to IDP '%x'\n", replyTo);
814
815     // Parse the description field to identify the datatype and schema, separated
            by ";"
816     vector<string> inf;
817     tokenizer<char_sep<';'> >::tokenize(inf, dataTypeSchema);
818     dataType = (char *) malloc(inf[0].size() * sizeof(char) + 1); //Hans Valgrind
            (+1 was missing)
819     strcpy(dataType, inf[0].c_str());
820     schema = (char *) malloc(inf[1].size() * sizeof(char) + 1); //Hans Valgrind (+1
            was missing)
821     strcpy(schema, inf[1].c_str());
822
```

```
823     if(startNewMerc(dataType, NULL, schema/*, newIDP*/) < 0) {
824       // Send an error message, if a replyTo label was specified
825       if(replyTo != 0) {
826         allocateXRPMsg(&replyMsg);
827         int totalSize, arglen;
828         totalSize = fillXRPCommand(replyMsg, XRP_CMD_ERROR);
829         arglen = addXRPArg(replyMsg, XRP_CLASS_NAME, dataTypeSchema, descLen);
830         if(arglen < 0) {
831           anaPrint(ANA_ERR, "Unable to add name to message\n");
832           free(replyMsg);
833           return;
834         }
835         totalSize += arglen;
836
837         anaL0_send(replyTo, replyMsg, totalSize);
838         free(replyMsg);
839       }
840     } else if(replyTo != 0) {
841       // Send back the same received message to publisher as success report
842
843       // First store the new data compartment into the metadata compartment
844       anaLabel_t templabel = mercIDPs[dataType];
845
846       // Find newly created router object so that we know what the joinlocation is
847       ANAMercuryNode *router = mercNodes[templabel];
848
849       MetadataEvent *ev = new MetadataEvent();
850       ev->SetSchema(schema);
851
852       joinLocations = (char *) malloc(1024 * sizeof(char));
853
854       vector<string> js;
855       tokenizer<comma_sep>::tokenize(js, schema);
856
857       // Go through all the attributes in the schema
858       for (uint32 i = 0; i < js.size(); i++) {
859         vector<string> inf;
860
861         // Identify lower and upper bounds
862         tokenizer<char_sep<':'> >::tokenize(inf, js[i]);
863         if(i > 0) {
864           sprintf(joinLocations, "%s,%s:%s", joinLocations, inf[0].c_str(), router
                ->GetHubManager()->GetAddress().ToString());
865         } else {
866           sprintf(joinLocations, "%s:%s", inf[0].c_str(), router->GetHubManager()->
                GetAddress().ToString());
867         }
868       }
869
870       ev->SetLocation(joinLocations);
871
872       // Creating metadata event
873       unsigned int minVal;
874       string dataTypeStr(dataType);
875       stringstream ss(dataTypeStr);
876       ss >> minVal;
877
878       Constraint c (0, minVal, minVal);
879       ev->AddConstraint (c);
880
881       // Make the data item permanent
882       ev->SetLifeTime(-1);
```

```
883
884       // Send it via the metadata (core) router
885       m_Router->SendEvent(ev);
886
887       // Send the reply message
888       anaPrint(ANA_DEBUG, "Sending publish scuccess to IDP '%x' Data compartment
              IDP is '%x'.\n", replyTo, templabel);
889
890       anaL0_send(replyTo, msg, len);
891
892       delete ev;
893     }
894
895     free(schema); // Hans Valgrind
896     free(dataType); // Hans Valgrind
897
898     return;
899   }
900
901   /* Remove data compartment. I.e. shut down MercuryNode */
902   void handleUnpublishCore(void *msg, int len) {
903     anaPrint(ANA_DEBUG, "UNPUBLISH - In metadata compartment\n");
904   /*
905
906     char *dataType = NULL;
907     char *labelChar = NULL;
908     string node_name = "mcis_";
909     int dataTypeLen = 0;
910     int ret = 0;
911     anaLabel_t node_label;
912     ANAMercuryNode *node = NULL;
913
914     // Resolve data type
915     dataType = (char *) getXRPArg( (char *) msg, XRP_CLASS_LKP_DESCR, &dataTypeLen,
            0);
916     if(dataType == NULL) {
917       anaPrint(ANA_ERR, "Unable to identify data type\n");
918       return;
919     }
920
921     node_name.append(dataType);
922
923     // Locate ANA Node
924     node_label = mercIDPs[dataType];
925     node = mercNodes[node_label];
926
927     anaPrint(ANA_NOTICE, "Shutting down node %s now!\n", node_name.c_str());
928     // node->UnregisterApplication(); node->Stop();
929     node->Shutdown();
930     // delete node; // MercuryNode destructor calls Stop()
931
932     memset(&g_Preferences, 0, sizeof(g_Preferences));
933
934     ret = anaL0_unregisterCallback(node_label, NULL);
935     if (ret < 0) {
936       anaPrint(ANA_ERR, "Unable to unregister callback IDP\n");
937     }
938
939     // Remove from maps
940     mercIDPs.erase(dataType);
941     mercNodes.erase(node_label);
942     mercSchemas.erase(node_label);
```

```
943
944     free(labelChar);
945   */
946     return;
947   }
948
949   /* Find data compartment */
950   void handleResolveCore(void *data, int len) {
951     anaPrint(ANA_NOTICE, "RESOLVE - In metadata compartment\n");
952
953     int dummy = 0;
954     char *dataType = NULL;
955     anaLabel_t replyTo = 0;
956     anaLabel_t result = 0;
957     anaLabel_t *labelP = NULL;
958
959     dataType = (char *) getXRPArg((xrpMsg_t) data, XRP_CLASS_DST_SRV, &dummy, 0);
960     if(dataType == NULL) {
961       anaPrint(ANA_ERR, "No data compartment name found.\n");
962       return;
963     } else {
964       anaPrint(ANA_DEBUG, "Resolving data compartment '%s'.\n", dataType);
965     }
966
967     /*
968     replyTo = (anaLabel_t) getXRPArg((xrpMsg_t) data, XRP_CLASS_LABEL, &dummy, 0);
969     if(replyTo == 0) {
970       anaPrint(ANA_ERR, "No label to reply to.\n");
971       return;
972     } else {
973       anaPrint(ANA_DEBUG, "Reply to IDP '%x'.\n", replyTo);
974     }
975
976     int labelSize = anaL1_decResponse((char *) data, XRP_CLASS_LABEL , 0, (void **)
           &labelP);
977     replyTo = *labelP;
978
979     anaPrint(ANA_DEBUG, "Reply to IDP '%x'.\n", replyTo);
980     */
981
982
983     char *client_ip = (char *) getXRPArg((xrpMsg_t) data, XRP_CLASS_MESSAGE, &dummy
           , 0);
984     anaPrint(ANA_DEBUG, "Client IP '%s'\n", client_ip);
985     char *client_service = (char *) getXRPArg((xrpMsg_t) data, XRP_CLASS_SRC_SRV, &
           dummy, 0);
986     anaPrint(ANA_DEBUG, "Client service '%s'\n", client_service);
987
988     replyTo = find_client(client_ip, client_service);
989     anaPrint(ANA_DEBUG, "Reply to IDP '%x'\n", replyTo);
990
991     result = mercIDPs[dataType];
992     if(result == 0) {
993       anaPrint(ANA_DEBUG, "Searching for data compartment '%s' in entire metadata
             space.\n", (char *) dataType);
994       // No cache (i.e. local fork) of the corresponding mercury node running,
             search in the metadata space
995       Query *q = new Query();
996
997   /*  XXX Hans Probably better version, supporting strings
998       MercuryID val((const char *) dataType);
999       Constraint c(0, val, val);
```

```
1000        q->AddConstraint(c);
1001    */
1002        unsigned int minVal;
1003        string dataTypeStr(dataType);
1004        stringstream ss(dataTypeStr);
1005        ss >> minVal;
1006        Constraint c(0, minVal, minVal);
1007        q->AddConstraint(c);
1008
1009
1010        m_Router->RegisterQuery(q);
1011
1012        // Check if we receive query results
1013        TimeVal v;
1014        gettimeofday(&v, NULL);
1015        uint64 nowtime = ((uint64) v.tv_sec * USEC_IN_SEC) + v.tv_usec;
1016        uint64 stopTime = nowtime + (unsigned long long) RESOLVE_TO * 2000000;
1017        MetadataEvent *e = NULL;
1018        while(e == NULL && nowtime < stopTime) {
1019          e = (MetadataEvent *) m_Router->ReadEvent();
1020          OS::SleepMillis(100);
1021          gettimeofday(&v, NULL);
1022          nowtime = ((uint64) v.tv_sec * USEC_IN_SEC) + v.tv_usec;
1023        }
1024
1025        delete q, c;
1026
1027        if(e == NULL) {
1028          // Didn't find such data record
1029          anaPrint(ANA_ERR, "Could not find the data compartment '%s' in the metadata
                space.\n", (char *) dataType);
1030          anaPrint(ANA_DEBUG, "Query='%s'\n", (char *) q);
1031          fillXRPCommand((xrpMsg_t) data, XRP_CMD_ERROR);
1032          anaL0_send(replyTo, data, len);
1033          return;
1034        }
1035
1036        // Found a data type from the metadata compartment. Start a local fork
              joining the other
1037        if(startNewMerc(dataType, (char *) e->GetLocation(), (char *) e->GetSchema()
              /*, result*/) < 0) {
1038          anaPrint(ANA_ERR, "Could not start a new data compartment node.\n");
1039          fillXRPCommand((char *) data, XRP_CMD_ERROR);
1040          anaL0_send(replyTo, data, len);
1041          return;
1042        } else {
1043          result = mercIDPs[dataType];
1044          anaPrint(ANA_DEBUG, "Started local peer to take part in data compartment.
                IDP to it is '%x'.\n", result);
1045        }
1046      } else {
1047        anaPrint(ANA_DEBUG, "Found data compartment in local cache. IDP to it is '%x
              '\n", result);
1048      }
1049
1050      // Sending resolve reply
1051      xrpMsg_t reply_msg = anaL1_allocateMessage();
1052      // REDUNDANT memset(reply_msg, 0, defaultXrpSpecs.maxMsgSize);
1053      // REDUNDANT allocateXRPMsg(&reply_msg);
1054      int total_size = fillXRPCommand(reply_msg, XRP_CMD_RESOLVE);
1055      if(total_size < 0) {
1056        anaPrint(ANA_ERR, "Unable to fill command in message\n");
```

```
1057      return;
1058    }
1059
1060    int arg_size = addXRPArg(reply_msg, XRP_CLASS_LABEL, &result, sizeof(anaLabel_t
          ));
1061    if(arg_size < 0) {
1062      anaPrint(ANA_ERR, "Unable to add label to resolve reply message\n");
1063      free(reply_msg);
1064      return;
1065    }
1066    total_size += arg_size;
1067
1068  /*
1069    arg_size = addXRPArg(reply_msg, XRP_CLASS_DST_SRV, &result, sizeof(anaLabel_t))
          ;
1070    if(arg_size < 0) {
1071      anaPrint(ANA_ERR, "Unable to add service to resolve reply message\n");
1072      free(reply_msg);
1073      return;
1074    }
1075    total_size += arg_size;
1076  */
1077
1078    anaPrint(ANA_DEBUG, "Sending resolve reply of size '%d' to IDP '%x'\n",
          total_size, replyTo);
1079    anaL0_send(replyTo, reply_msg, total_size);
1080    free(reply_msg);
1081
1082    return;
1083  }
1084
1085  void handleLookupCore(void *argus) {
1086    anaPrint(ANA_NOTICE, "LOOKUP - In metadata compartment\n");
1087  /*
1088    int targetLen, contextLen,labelSize, descLen, ret, dummy, i = 0;
1089    char *dataType;
1090    char *query;
1091    anaLabel_t replyTo, copyReplyTo, result, input;
1092    void *msg;
1093    int len;
1094    //struct anaMinmexSpecs_s *minmex;
1095    struct LookupArgs * _argus = (struct LookupArgs *) argus;
1096    input = _argus->input;
1097    msg = _argus->data;
1098    len = _argus->len;
1099    //minmex = _argus->originMinmex;
1100
1101    replyTo = (anaLabel_t) getXRPArg((char *) msg, XRP_CLASS_LABEL, &labelSize, 0);
1102    if (replyTo == 0) {
1103      anaPrint(ANA_ERR, "Unable to find reply label\n");
1104      return;
1105    }
1106
1107    query = (char *) getXRPArg((char *) msg, XRP_CLASS_DST_CXT, &targetLen, 0); //
          XRP_CLASS_TARGET
1108    if (query == 0) {
1109      return;
1110    }
1111
1112    // Search directly in the metadata space
1113    MercuryID val((const char*)query);
1114    Constraint c (0, val, val);
```

```
1115    //THIS PREFIX STUFF MUST BE TESTED, UNSURE!!!
1116    string str(query);
1117    if (str.find("*")!=string::npos) {
1118      c.SetPrefix();
1119    }
1120    Query *q = new Query();
1121    q->AddConstraint(c);
1122    m_Router->RegisterQuery(q);
1123    delete q,c;
1124
1125    // Check if we receive query results
1126    MetadataEvent *e = NULL;
1127    int totalLength = 0;
1128    int arglen = 0;
1129    TimeVal v;
1130    gettimeofday(&v, NULL);
1131    uint64 nowtime = ((uint64)v.tv_sec*USEC_IN_SEC)+v.tv_usec;
1132    uint64 stopTime = nowtime + (unsigned long long) LOOKUP_TO * 1000000;
1133    while(nowtime < stopTime) {
1134      e = (MetadataEvent *) m_Router->ReadEvent();
1135      if(e != NULL) {
1136        anaPrint(ANA_NOTICE, "Lookup returned '%s'", (char *) e);
1137        // Send result tuple back to the querier
1138        memset(msg, 0, defaultXrpSpecs.maxMsgSize);
1139        totalLength = fillXRPCommand((char *) msg, XRP_CMD_DATA);
1140        if (totalLength < 0) {
1141          anaPrint(ANA_ERR, "Unable to fill command in message\n");
1142          return;
1143        }
1144        char *minStr = (char *) malloc(100 * sizeof(char));
1145        sprintf(minStr,"%ld", e->GetConstraint(0)->GetMin().getsi());
1146        arglen = addXRPArg((char *) msg, XRP_CLASS_LKP_DESCR, minStr, sizeof(minStr
               ));
1147        totalLength += arglen;
1148        anaL0_send(replyTo, msg, totalLength);
1149      }
1150      OS::SleepMillis(100);
1151      gettimeofday(&v, NULL);
1152      nowtime = ((uint64) v.tv_sec * USEC_IN_SEC) + v.tv_usec;
1153    }
1154  */
1155
1156    return;
1157  }
1158
1159  //-DEFAULT BRICK FUNCTIONS-----------------------------------------------------------
1160  void brick_exit() {
1161    anaPrint(ANA_DEBUG, "MCIS EXIT\n");
1162  /*
1163    xrpMsg_t msg = anaL1_allocateMessage();
1164    int ret = 0;
1165    int len = 0;
1166
1167    anatimer_exit();
1168
1169    len = anaL1_encCompartmentUnpublish(msg, coreInput, &context, &service);
1170
1171    ret = anaL1_requestReply(myNodeCompLabel, msg, len, (anaCallback_t) &
           handleUnpublishReply, NULL, NULL, NULL, NOTHREAD);
1172    if(ret < 0) { anaPrint(ANA_ERR, "Unable to unpublish MCIS\n"); }
1173    free(msg);
1174
```

```
1175    for(mercIDPsIter = mercIDPs.begin(); mercIDPsIter != mercIDPs.end();
            mercIDPsIter++ ) { delete mercIDPsIter->second; }
1176    for(mercNodesIter = mercNodes.begin(); mercNodesIter != mercNodes.end();
            mercNodesIter++ ) { delete mercNodesIter->second; }
1177    for(mercSchemasIter = mercSchemas.begin(); mercSchemasIter != mercSchemas.end()
            ; mercSchemasIter++ ) { delete mercSchemasIter->second; }
1178
1179    mercIDPs.clear();
1180    mercNodes.clear();
1181    mercSchemas.clear();
1182  */
1183  }
1184
1185  /* The brick_start function is the main function of each brick */
1186  int brick_start() {
1187    int ret = 0;
1188    int lenMsg, labelSize;
1189    xrpMsg_t msg = anaL1_allocateMessage();
1190    char *publ_name;
1191    pubNameExt = getAuxArg(0);
1192    int i = 0;
1193
1194    anaL2_initDefault();
1195
1196    /* Signal handler for resource adaptation */
1197    if(signal(SIGUSR1, sigusr1_handler) == SIG_ERR) {
1198      anaPrint(ANA_ERR, "Could not start signal handler\n");
1199    } else {
1200      anaPrint(ANA_DEBUG, "Signal handler for increasing resource consumption in
             place. Signal is '%d'\n", SIGUSR1);
1201    }
1202
1203    /* Using signal 31 instead of SIGUSR2 because Mercury threads are using it */
1204    if(signal(31, sigusr2_handler) == SIG_ERR) {
1205      anaPrint(ANA_ERR, "Could not start signal handler\n");
1206    } else {
1207      anaPrint(ANA_DEBUG, "Signal handler for decreasing resource consumption in
             place. Signal is '%d'\n", 31);
1208    }
1209
1210    // get the first external arg to designate name extension to the publish name
           in the node
1211    // needed to distinguish two MCIS bricks by the test brick
1212    char **argVector = (char **) malloc(100 * sizeof(char *));
1213
1214    char *tmpArg = getAuxArg(i); // CHANGE BACK TO i-1 AFTER TESTING!!!!
1215    while(tmpArg != NULL) {
1216      argVector[i] = (char *) malloc(strlen(tmpArg) + 1);
1217      strcpy(argVector[i], tmpArg);
1218      tmpArg = getAuxArg(i++); // CHANGE BACK TO i++ AFTER TESTING!!!!
1219    }
1220
1221    int argCount = i;
1222
1223    InitializeMercury(&argCount, argVector, NULL, true);
1224    free(argVector);
1225
1226    DBG_INIT (NULL);
1227
1228    // Start the "core" MercuryNode which is part of the metadata compartment that
           stores all the data compartments
1229    string MNname = string("coremcis_") + string(g_Preferences.hostname);
```

```
1230    m_Router = ANAMercuryNode::GetInstance(MNname.c_str());
1231    m_Router->SetID(MNname.c_str());
1232
1233    // RAM Storage for 1 hub. I.e. the hub responsible for data compartments.
1234    anaPrint(ANA_DEBUG, "Creating data store for data compartments\n");
1235    DummyApp *m_app = new DummyApp(1);
1236    m_Router->RegisterApplication(m_app);
1237    m_Router->FireUp();
1238
1239    i = 0;
1240    // Send a ping to get alljoined status (needed when joining late or when
          joining a peer (not bootstrap))
1241    if(g_Preferences.join_locations[0]) {
1242      // Join peer
1243      anaPrint(ANA_DEBUG, "Joining peer\n");
1244      vector<string> js;
1245      tokenizer<comma_sep>::tokenize(js, g_Preferences.join_locations);
1246      //for (uint32 i = 0; i < js.size(); i++) { only consider the first attr hub
1247      vector<string> inf;
1248      tokenizer<char_sep<':'> >::tokenize(inf, js[0]);
1249      string joinId = inf[1];
1250      while (m_Router->SendPing (EndPoint((char *) joinId.c_str())) == -1) {
1251        #ifndef ENABLE_ANANET_THREAD
1252          ANANet::DoWork (100);
1253        #else
1254          OS::SleepMillis(100); //we are using a thread
1255        #endif
1256        #ifndef HAVE_THREADS
1257          m_Router->DoWork (200);
1258        #else
1259          OS::SleepMillis(200); //we are using threads
1260        #endif
1261      }
1262    } else if (g_Preferences.bootstrap != NULL) {
1263      // Join bootstrap server
1264      anaPrint(ANA_DEBUG, "Joining bootstrap server\n");
1265
1266      while (m_Router->SendPing(g_Preferences.bootstrap) == -1) {
1267        #ifndef ENABLE_ANANET_THREAD
1268          ANANet::DoWork (100);
1269        #else
1270          OS::SleepMillis(100); //we are using a thread
1271        #endif
1272        #ifndef HAVE_THREADS
1273          m_Router->DoWork (200);
1274        #else
1275          OS::SleepMillis(200); //we are using threads
1276        #endif
1277      }
1278    } else {
1279      // this is the case when we need to start searching for another running MCIS
            brick, locally (node cmpt) or remotely (ip/eth cmpt)
1280      //FILL HERE LATER
1281    }
1282
1283    while (!m_Router->AllJoined ()) {
1284      #ifndef ENABLE_ANANET_THREAD
1285        ANANet::DoWork (100);
1286      #else
1287        OS::SleepMillis(100); //we are using a thread
1288      #endif
1289      #ifndef HAVE_THREADS
```

```
1290        m_Router->DoWork (200);
1291      #else
1292        OS::SleepMillis(200); //we are using threads
1293      #endif
1294    }
1295
1296    anaPrint(ANA_NOTICE, "JOINED ALL HUBS!\n");
1297
1298    // Publish callback function for meta data compartment
1299    publ_name = (char *) malloc(100 * sizeof(char));
1300    sprintf(publ_name, "MCIS_%s+compartment+mcis", pubNameExt);
1301
1302    memset (&service, 0, sizeof(struct service_s));
1303    memset (&context, 0, sizeof(struct context_s));
1304    memset (&timeout, 0, sizeof(struct timespec));
1305
1306    context.value = (void *) "*";
1307    context.valueLen = 2;
1308
1309    service.value = publ_name;
1310    service.valueLen = strlen(publ_name) + 1;
1311
1312    timeout.tv_sec = 9;
1313
1314    service_ip.value = (void *) "ip";
1315    service_ip.valueLen = strlen((const char*) service_ip.value) + 1;
1316
1317    ipIDP = anaL2_resolve(NODE_LABEL, &context, &service_ip, 'u', NULL, &timeout);
1318    if (ipIDP == 0) {
1319      anaPrint(ANA_ERR, "Can't find IP compartment\n");
1320      return -1;
1321    }
1322
1323    coreInput = anaL2_publish(ipIDP, &context, &service, (AL2Callback_t) &
         recvFromHL, THREAD, &timeout);
1324    if (coreInput == 0) {
1325      anaPrint(ANA_ERR, "Unable to publish '%s' in IP compartment.\n", (char *)
           service.value);
1326      return -1;
1327    } else {
1328      anaPrint(ANA_DEBUG, "Published '%s' in IP compartment as IDP '%x'\n", (char
           *) service.value, coreInput);
1329    }
1330
1331    anaPrint(ANA_NOTICE, "Starting infinite loop.\n");
1332    #ifdef ENABLE_ANANET_THREAD
1333      #ifdef HAVE_THREADS
1334        return 1;
1335      #endif
1336    #endif
1337
1338    while (true) {
1339      #ifndef ENABLE_ANANET_THREAD
1340        ANANet::DoWork (20);
1341      #else
1342        OS::SleepMillis(20);
1343      #endif
1344      #ifndef HAVE_THREADS
1345        m_Router->DoWork (20);
1346      #else
1347        OS::SleepMillis(20);
1348      #endif
```

```
1349    }
1350
1351    return 1;
1352  }
```

Listing 8: MCIS Peer

B.5 mcis_benchmarker.cpp

```
1   /*------------------------- Default brick header code -----------------------------*/
2   /* brick_template.h contains the functions that initialize the bricks */
3   extern "C" {
4     #include "brick_template.h"
5   }
6   char *mymodename = (char *) "mcis_benchmarker";
7   #ifdef KERNEL
8     MODULE_LICENSE("Dual BSD/GPL");
9     MODULE_AUTHOR("ANA Project");
10  #endif
11  /*------------------------- End default brick header code ---------------------*/
12
13  /*------------------------- Brick specific code starts here-------------------*/
14
15  // Standard includes
16  #include <iostream>
17  #include <fstream>
18  #include <sstream>
19  #include <stdio.h>
20  #include <stdlib.h>
21  #include <unistd.h>
22
23  using namespace std;
24
25  anaLabel_t ipIDP = 0; // IP compartment IDP
26  anaLabel_t myIDP = 0; // Own IDP in IP compartment
27  anaLabel_t dataIDP = 0; // Data compartment IDP in IP compartment
28  anaLabel_t metadataIDP = 0; // Metadata compartment IDP in IP compartment
29
30  char *mcisip = NULL; // ANA IP where MCIS is located
31  char *myip = NULL; // ANA IP of this node
32
33  int test_ipbrick() {
34    anaPrint(ANA_DEBUG, "INFO: Locating IP compartment\n");
35    struct service_s service;
36    struct context_s context;
37    struct timespec timeout;
38
39    memset(&service, 0, sizeof(struct service_s));
40    memset(&context, 0, sizeof(struct context_s));
41    memset(&timeout, 0, sizeof(struct timespec));
42
43    timeout.tv_sec = 9;
44
45    context.value = (void *) "*";
46    context.valueLen = 2;
47
48    service.value = (void *) "ip";
49    service.valueLen = strlen((const char *) service.value) + 1;
50
51    ipIDP = anaL2_resolve(NODE_LABEL, &context, &service, 'u', NULL, &timeout);
```

```
52    if(ipIDP == 0) {
53      anaPrint(ANA_ERR, "Can't find IP compartment\n");
54      return -1;
55    } else {
56      anaPrint(ANA_DEBUG, "INFO: Discovered IP brick at IDP '%x'\n", ipIDP);
57    }
58
59    /* Make this brick available in IP compartment for feedback from MCIS */
60    struct service_s service_2;
61    memset (&service_2, 0, sizeof(struct service_s));
62    service_2.value = (void *) mymodename;
63    service_2.valueLen = strlen((const char*) service_2.value) + 1;
64    myIDP = anaL2_publish(ipIDP, &context, &service_2, NULL, NOTHREAD, &timeout);
65    if(myIDP == 0) {
66      anaPrint(ANA_ERR, "Unable to publish '%s' in IP compartment.\n", (char *)
          service_2.value);
67      return -1;
68    } else {
69      anaPrint(ANA_DEBUG, "Published '%s' in IP compartment as IDP '%x'\n", (char
          *) service_2.value, myIDP);
70    }
71
72    return 1;
73  }
74
75  /* Resolve metadata compartment from IP compartment */
76  int test_mcis() {
77    anaPrint(ANA_DEBUG, "INFO: Locating MCIS metadata compartment\n");
78    struct service_s metadata_service;
79    struct context_s metadata_context;
80    struct timespec timeout;
81
82    memset (&metadata_service, 0, sizeof(struct service_s));
83    memset (&metadata_context, 0, sizeof(struct context_s));
84    memset (&timeout, 0, sizeof(struct timespec));
85
86    metadata_service.value = (char *) "mcis";
87    metadata_service.valueLen = strlen((const char*) metadata_service.value) + 1;
88
89    metadata_context.value = (char *) mcisip;
90    metadata_context.valueLen = strlen((const char*) metadata_context.value) + 1;
91
92    timeout.tv_sec = 20;
93
94    metadataIDP = anaL2_resolve(ipIDP, &metadata_context, &metadata_service, 'u',
          NULL, &timeout);
95    if(metadataIDP == 0) {
96      anaPrint(ANA_ERR, "ERROR: Discovering metadata compartment at %s failed\n",
          mcisip);
97      return -1;
98    } else {
99      anaPrint(ANA_DEBUG, "INFO: Discovered metadata compartment at IDP '%x'.\n",
          metadataIDP);
100   }
101
102   return 1;
103 }
104
105 /* Publish in metadata compartment */
106 int test_publish_schema(char *schema) {
107   anaPrint(ANA_NOTICE, "INFO: Sending schema publish message to metadata
          compartment\n");
```

```
108
109     struct service_s service;
110     struct context_s context;
111     struct timespec timeout;
112
113     memset (&service, 0, sizeof(struct service_s));
114     memset (&context, 0, sizeof(struct context_s));
115     memset (&timeout, 0, sizeof(struct timespec));
116
117     service.value = schema;
118     service.valueLen = strlen((const char*) service.value) + 1;
119
120     context.value = (char *) "*";
121     context.valueLen = strlen((const char*) context.value) + 1;
122
123     timeout.tv_sec = 30;
124
125     xrpMsg_t msg = anaL1_allocateMessage();
126     int msg_length = anaL1_encCompartmentPublish(msg, myIDP, &context, &service,
          IDP_PERM);
127     if(msg_length < 0 ) {
128       anaPrint(ANA_ERR, "ERROR: Unable to encode schema publish message\n");
129       return -1;
130     }
131
132     /* Add our description so MCIS can resolve and respond */
133     int arg_length = addXRPArg(msg, XRP_CLASS_SRC_SRV, mymodename, strlen(
          mymodename) + 1);
134     if(arg_length < 0) {
135       anaPrint(ANA_ERR, "ERROR: Can't add my description to schema publish message\
            n");
136       return -1;
137     }
138     msg_length += arg_length;
139
140     /* Add our IP so MCIS can resolve and respond */
141     arg_length = addXRPArg(msg, XRP_CLASS_MESSAGE, myip, strlen(myip) + 1);
142     if(arg_length < 0) {
143       anaPrint(ANA_ERR, "ERROR: Can't add my IP to schema publish message\n");
144       return -1;
145     }
146     msg_length += arg_length;
147
148
149     /* Send message */
150     int ret = anaL0_send(metadataIDP, msg, msg_length);
151     if(ret < 0) {
152       anaPrint(ANA_ERR, "ERROR: Unable to send schema publish message to metadata
            compartment\n");
153       return -1;
154     }
155
156     struct anaL2_message *reply_msg = anaL2_receive(myIDP, &timeout);
157     if(anaL1_isError((char *) reply_msg)) {
158       anaPrint(ANA_ERR, "ERROR: Got error back from metadata compartment. Unable to
             publish schema\n");
159       return -1;
160     } else {
161       anaPrint(ANA_NOTICE, "INFO: Schema published in metadata compartment\n");
162     }
163
164     return 1;
```

```
165 }
166
167 /* Resolve in metadata compartment */
168 int test_data_compartment(char *schema) {
169   int dummy = 0;
170   char *cmp_name = NULL;
171   cmp_name = strtok(schema, ";");
172
173   struct service_s service;
174   struct context_s context;
175   struct timespec timeout;
176
177   memset (&service, 0, sizeof(struct service_s));
178   memset (&context, 0, sizeof(struct context_s));
179   memset (&timeout, 0, sizeof(struct timespec));
180
181   anaPrint(ANA_NOTICE, "Resolving data compartment='%s'\n", cmp_name);
182
183   context.value = (char *) "*";
184   context.valueLen = strlen((const char*) context.value) + 1;
185
186   service.value = (char *) cmp_name;
187   service.valueLen = strlen((const char*) service.value) + 1;
188
189   timeout.tv_sec = 20;
190
191   // Preparing message
192   xrpMsg_t msg = anaL1_allocateMessage();
193   int msg_length = anaL1_encResolve(msg, &context, &service, 'u', NULL);
194   if(msg_length < 0 ) {
195     free(msg);
196     anaPrint(ANA_ERR, "ERROR: Unable to encode resolve data compartment message\n
          ");
197     return -1;
198   }
199
200   // Add description so MCIS can resolve and respond
201   int arg_length = addXRPArg(msg, XRP_CLASS_SRC_SRV, mymodename, strlen(
        mymodename) + 1);
202   if(arg_length < 0) {
203     anaPrint(ANA_ERR, "ERROR: Can't add my description to schema publish message\
          n");
204     return -1;
205   }
206   msg_length += arg_length;
207
208   // Add our IP so MCIS can resolve and respond
209   arg_length = addXRPArg(msg, XRP_CLASS_MESSAGE, myip, strlen(myip) + 1);
210   if(arg_length < 0) {
211     anaPrint(ANA_ERR, "ERROR: Can't add my IP to schema publish message\n");
212     return -1;
213   }
214   msg_length += arg_length;
215
216   // Sending resolve message
217   int ret = anaL0_send(metadataIDP, msg, msg_length);
218   if(ret < 0) {
219     anaPrint(ANA_ERR, "ERROR: Unable to send the resolve data compartment message
          to metadata compartment\n");
220     free(msg);
221     return -1;
222   }
```

```
223
224     // Expecting feedback from metadata compartment
225     struct anaL2_message *reply_msg = anaL2_receive(myIDP, &timeout);
226     if(anaL1_isError((char *) reply_msg)) {
227       anaPrint(ANA_ERR, "ERROR: Got error back from metadata compartment.\n");
228       return -1;
229     } else {
230       anaPrint(ANA_NOTICE, "INFO: Got feedback from metadata compartment. A
              corresponding data compartment is ready\n");
231     }
232
233     // Resolving data compartment
234     struct service_s data_service;
235     struct context_s data_context;
236     memset(&data_service, 0, sizeof(struct service_s));
237     memset(&data_context, 0, sizeof(struct service_s));
238
239     data_context.value = (void *) mcisip;
240     data_context.valueLen = strlen((const char*) data_context.value) + 1;
241
242     string data_compartment = "compartment+mcis_data_compartment+" + string(
          cmp_name);
243     data_service.value = (char *) data_compartment.c_str();
244     data_service.valueLen = strlen((const char*) data_service.value) + 1;
245
246     dataIDP = anaL2_resolve(ipIDP, &data_context, &data_service, 'u', NULL, &
          timeout);
247
248     anaPrint(ANA_NOTICE, "INFO: Data compartment found at IDP '%x'.\n", dataIDP);
249
250     return 1;
251   }
252
253   /* Publish in data compartment */
254   int test_save_data(char *flowfilename, int data_elements) {
255     anaPrint(ANA_NOTICE, "INFO: Publishing data in the data compartment\n");
256
257     char flow[1024];
258     int i = 0;
259     ifstream flowifs(flowfilename, ifstream::in);
260
261     struct service_s service;
262     struct context_s context;
263     struct timespec timeout;
264     memset(&context, 0, sizeof(struct context_s));
265     memset(&timeout, 0, sizeof(struct timespec));
266
267     context.value = (char *) "*";
268     context.valueLen = 2;
269     timeout.tv_sec = 9;
270
271     // Iterate trough flow file and publish data, line by line
272     while(flowifs.good() && i <= data_elements) {
273       // Each line is comma separated values
274       flowifs.getline(flow, 1024);
275
276       // Make publish message
277       memset(&service, 0, sizeof(struct service_s));
278       service.value = (char *) flow;
279       service.valueLen = strlen((const char *) service.value) + 1;
280
```

```
281       anaPrint(ANA_DEBUG, "Saving data element '%d': '%s'\n", i, (char *) service.
              value);
282
283       // Perpare message
284       xrpMsg_t msg = anaL1_allocateMessage();
285       int msg_length = anaL1_encCompartmentPublish(msg, myIDP, &context, &service,
              IDP_PERM);
286       if(msg_length < 0 ) {
287         anaPrint(ANA_ERR, "ERROR: Unable to encode publish data item message\n");
288       }
289
290       // Add our description so MCIS can resolve and respond
291       int arg_length = addXRPArg(msg, XRP_CLASS_SRC_SRV, mymodename, strlen(
              mymodename) + 1);
292       if(arg_length < 0) {
293         anaPrint(ANA_ERR, "ERROR: Can't add my description to data item publish
                message\n");
294       }
295       msg_length += arg_length;
296
297       // Add our IP so MCIS can resolve and respond
298       arg_length = addXRPArg(msg, XRP_CLASS_MESSAGE, myip, strlen(myip) + 1);
299       if(arg_length < 0) {
300         anaPrint(ANA_ERR, "ERROR: Can't add my IP to data item publish message\n");
301       }
302       msg_length += arg_length;
303
304       // Send publish message
305       int ret = anaL0_send(dataIDP, msg, msg_length);
306       if(ret < 0) {
307         anaPrint(ANA_ERR, "ERROR: Unable to send data item publish message to the
                data compartment\n");
308         free(msg);
309         i++;
310         continue;
311       }
312
313       // Check that the data compartment responds ands says the data is saved
314       struct anaL2_message *reply_msg = anaL2_receive(myIDP, &timeout);
315       if(anaL1_isError((char *) reply_msg)) {
316         anaPrint(ANA_ERR, "ERROR: Got error back from data compartment. Data item
                '%d' not stored\n", i);
317       } else {
318         anaPrint(ANA_DEBUG, "INFO: Data item '%d' stored\n", i);
319       }
320
321       free(msg);
322       i++;
323       sleep(0.4);
324     }
325
326     flowifs.close();
327
328     anaPrint(ANA_NOTICE, "Done publishing data\n\n");
329
330     return 1;
331   }
332
333   /* Lookup in data compartment */
334   void test_query(void *query) {
335     anaPrint(ANA_NOTICE, "Making query='%s'\n", (char *) query);
336
```

```
337     struct service_s query_service;
338     struct context_s query_context;
339     struct timespec query_timeout;
340     struct service_s our_service;
341
342     memset(&query_service, 0, sizeof(struct service_s));
343     memset(&query_context, 0, sizeof(struct context_s));
344     memset(&query_timeout, 0, sizeof(struct timespec));
345     memset(&our_service, 0, sizeof(struct service_s));
346
347     query_context.value = (char *) "*";
348     query_context.valueLen = 2;
349
350     query_service.value = (char *) query;
351     query_service.valueLen = strlen((const char *) query_service.value) + 1;
352
353     // Add our description so MCIS can resolve and respond
354     our_service.value = (char *) mymodename;
355     our_service.valueLen = strlen((const char *) our_service.value) + 1;
356
357     // Set timeout to greater than 10 because 10 is the timout MCIS uses internally
358     query_timeout.tv_sec = 20;
359
360     // Perpare message
361     xrpMsg_t msg = anaL1_allocateMessage();
362     int msg_length = anaL1_encLookup(msg, &query_context, &query_service, &
            our_service);
363     if(msg_length < 0 ) {
364       anaPrint(ANA_ERR, "ERROR: Unable to encode query message\n");
365     }
366
367     // Add our IP so MCIS can resolve and respond
368     int arg_length = addXRPArg(msg, XRP_CLASS_MESSAGE, myip, strlen(myip) + 1);
369     if(arg_length < 0) {
370       anaPrint(ANA_ERR, "ERROR: Can't add my IP to query message\n");
371     }
372     msg_length += arg_length;
373
374     int ret = anaL0_send(dataIDP, msg, msg_length);
375     if(ret < 0) {
376       anaPrint(ANA_ERR, "ERROR: Unable to send query message to data compartment 1\
            n");
377     }
378
379     free(msg);
380
381     // Check that the data compartment responds
382     struct anaL2_message *reply_msg = anaL2_receive(myIDP, &query_timeout);
383     if(anaL1_isError((char *) reply_msg)) {
384       anaPrint(ANA_ERR, "ERROR: Got error back from data compartment. Query failed\
            n");
385     } else {
386       // Go trough results and print them
387       char *result_string = NULL;
388
389       int i = 0;
390       while(anaL1_decResponse((char *) reply_msg->data, XRP_CLASS_LKP_DESCR, i, (
            void**) &result_string) > 0) {
391         anaPrint(ANA_NOTICE,"Query result: '%s'\n", result_string);
392         result_string = NULL ;
393         i++;
394       }
```

```
395     }
396
397     anaPrint(ANA_NOTICE, "Query compete.\n\n");
398   }
399
400   void brick_exit() {
401     anaPrint(ANA_NOTICE, "MCIS Benchmark brick finished!\n");
402   }
403
404   int brick_start() {
405     int ret = 0;
406     int queries_per_minute = 0;
407     int data_elements = 0;
408     char *schema = NULL;
409     char *datafilename = NULL;
410     char *query = NULL;
411
412     anaL2_initDefault();
413
414     if (getAuxArg(5) == NULL) {
415       printf("Usage: mcis_benchmarker -a <MCIS IP> -a <OUR IP> -a <SCHEMA> -a <DATA
              FILE> -a <DATA ELEMENTS> -a <QUERY> -a <QUERIES PER MINUTE>\n");
416       anaL0_stopANA(-1);
417       return -1;
418     } else {
419       mcisip = getAuxArg(0);
420       myip = getAuxArg(1);
421       schema = getAuxArg(2);
422       datafilename = getAuxArg(3);
423       data_elements = atoi(getAuxArg(4));
424       query = getAuxArg(5);
425       queries_per_minute = atoi(getAuxArg(6));
426     }
427
428     // Do all the tests
429     printf("Locating IP compartment");
430     ret = test_ipbrick();
431     if(ret <= 0) {
432       printf("An error occured. Quiting!\n");
433       anaL0_stopANA(-1);
434     }
435
436     printf(" and locating MCIS in IP compartment at '%s'\n", mcisip);
437     ret = test_mcis();
438     if(ret <= 0) {
439       printf("An error occured. Quiting!\n");
440       anaL0_stopANA(-1);
441     }
442
443     printf("Creating data structure\n");
444     ret = test_publish_schema(schema);
445     if(ret <= 0) {
446       printf("An error occured. Quiting!\n");
447       anaL0_stopANA(-1);
448     }
449
450     printf("Locating data compartment\n");
451     ret = test_data_compartment(schema);
452     if(ret <= 0) {
453       printf("An error occured. Quiting!\n");
454       anaL0_stopANA(-1);
455     }
```

```
456
457     printf("Storing '%d' data elements from '%s' in MCIS\n", data_elements,
            datafilename);
458     anaPrint(ANA_NOTICE, "!STORING.\n");
459     ret = test_save_data(datafilename, data_elements);
460     if(ret <= 0) {
461       printf("An error occured. Quiting!\n");
462       anaL0_stopANA(-1);
463     }
464
465     printf("Pause. This is a good place to start CPU demanding programs.\n");
466     sleep(2);sleep(2);sleep(2);sleep(2);sleep(2);
467
468     printf("Continous quering %d times per minute...\n", queries_per_minute);
469     anaPrint(ANA_NOTICE, "!QUERING. Query='%s' Size='%d'\n", query, strlen(query));
470     anatimer_init();
471     int interval = 1000 / ((double) queries_per_minute / 60);
472     anatimer_add(interval, ANATIMER_PERIODIC, ANATIMER_ABSOLUTE, test_query, query,
            strlen(query));
473
474     return 0;
475   }
```

Listing 9: MCIS Benchmarker

C Miscellaneous code

The following subsections are dedicated to utility programs needed in order to repeat our experiments.

C.1 ip2long.c

```
1  #include <arpa/inet.h>
2  #include <stdio.h>
3  #include <stdlib.h>
4  #include <string.h>
5
6  int main(int argc, char *argv[]) {
7    struct in_addr addr;
8    int ret = 0;
9    char *ip;
10
11   while(scanf("%s", ip) != EOF )  {
12     ret = inet_pton(AF_INET, ip, &addr);
13     if(ret <= 0) {
14       fprintf(stderr, "Not valid IPv4 address\n");
15       exit(EXIT_FAILURE);
16     }
17
18     printf("%lu\n", (unsigned long) addr.s_addr);
19   }
20
21   exit(EXIT_SUCCESS);
22 }
```

Listing 10: IP2Long

C.2 generate_cpu.c

```
1  #include <stdlib.h>
2  #include <stdio.h>
3
4  #define EVER ;;
5
6  int main(int argc, char **argv[]) {
7
8    for(EVER) {
9    }
10
11   exit(EXIT_SUCCESS);
12 }
```

Listing 11: Generate CPU utilization

D Shell scripts

These bash scripts are helpful for starting the different bricks used in this thesis and are made available for completness.

D.1 start_anaip.sh

```
1  #!/bin/bash
2  # Hans Vatne Hansen (hansvh@ifi.uio.no)
3
4  #-----CHANGE VARIABLES BELOW----------------------------------------------------------
5  ANA_BASE_DIR='/home/hansvh/dmms/ana/ana-core/devel'
6  IP=`/sbin/ifconfig | grep -m 1 "inet addr:" | awk -F: '{ print $2 }' | awk -F\
       '{ print $1 }'`
7  VLINKID='1234'
8  INTERFACE='eth1'
9  NETMASK='255.255.255.0'
10 #-----CHANGE VARIABLES ABOVE----------------------------------------------------------
11
12 echo "Starting Minmex..."
13 sudo $ANA_BASE_DIR/bin/minmex &
14 sleep 0.2
15
16 echo "Loading gates plugin..."
17 $ANA_BASE_DIR/bin/mxconfig load brick $ANA_BASE_DIR/so/gatesPlug.so
18 sleep 0.1
19 GATE=`ls --sort=time /tmp/anaControl_gatesPlug_* | head -n 1`
20
21 echo "Setting up network..."
22 $ANA_BASE_DIR/bin/mxconfig load brick $ANA_BASE_DIR/so/vlink.so
23 sleep 0.2
24 $ANA_BASE_DIR/bin/vlconfig create $VLINKID
25 $ANA_BASE_DIR/bin/vlconfig add_if vlink1 $INTERFACE
26 $ANA_BASE_DIR/bin/vlconfig up vlink1
27
28 $ANA_BASE_DIR/bin/mxconfig load brick $ANA_BASE_DIR/so/eth-vl.so
29 sleep 0.1
30 $ANA_BASE_DIR/bin/mxconfig load brick $ANA_BASE_DIR/so/ip_enc.so
31 sleep 0.1
32 $ANA_BASE_DIR/bin/mxconfig load brick $ANA_BASE_DIR/so/ip_fwd.so
33 sleep 0.1
34
35 sudo LD_LIBRARY_PATH=$ANA_BASE_DIR/lib/ $ANA_BASE_DIR/bin/ip_cfg -n unix://$GATE
       -i $IP -m $NETMASK -e eth01 &
36 sleep 2
37
38 $ANA_BASE_DIR/bin/mxconfig show bricks
```

Listing 12: Start MinMex and IP bricks

D.2 start_mcis_bootstrap.sh

```
1  #!/bin/bash
2  # Hans Vatne Hansen (hansvh@ifi.uio.no)
3
4  if [ -n "$1" ]
5  then
6     echo "Using $1 nodes..."
7  else
8     echo "USAGE: $0 [number of MCIS nodes]"
9     exit 1
10 fi
11
12 #-----CHANGE VARIABLES BELOW----------------------------------------------------------
13 TRANSPORT="ANAIP"
14 IDENTIFICATOR="dmms"
15 PREFIX="/home/hansvh/dmms/ana/ana-core/devel"
16 SCHEMA=$PREFIX"/C/bricks/mcis/mercury/configs/schema_mcis.cfg"
17 LOG_DIR="/home/hansvh/dmms/logs"
18 VERBOSITY="20"
19 DEBUGLEVEL="ANA_ALL"
20 #-----CHANGE VARIABLES ABOVE----------------------------------------------------------
21
22 NODES=$1
23 CONTROL_GATE=`ls --sort=time /tmp/anaControl_gatesPlug_* | head -n 1`
24
25 echo "Starting MCIS Bootstrap brick..."
26 sudo LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PREFIX/lib/:$PREFIX/C/bricks/mcis/mercury/
       $PREFIX/bin/mcis_bootstrap -D $DEBUGLEVEL -n unix://$CONTROL_GATE -a "--
      schema" -a $SCHEMA -a "--nservers" -a $NODES -a "--log-dir" -a $LOG_DIR -a "
      --merctrans" -a $TRANSPORT -a "--mercctrl" -a $TRANSPORT -a "-H" -a
      $IDENTIFICATOR -a "-v" -a $VERBOSITY -a "--sampling" -a "1" -a "--measurement
      " -a "1"
```

Listing 13: Start MCIS Bootstrap brick

D.3 start_mcis_peer.sh

```
1  #!/bin/bash
2  # Hans Vatne Hansen (hansvh@ifi.uio.no)
3
4  if [ -n "$1" ]
5  then
6     echo "Using $1 as MCIS ID..."
7  else
8     echo "USAGE: $0 [MCIS NR]"
9     exit 1
10 fi
11
12 #-----CHANGE VARIABLES BELOW----------------------------------------------------------
13 TRANSPORT="ANAIP"
14 PREFIX="/home/hansvh/dmms/ana/ana-core/devel"
15 SCHEMA=$PREFIX"/C/bricks/mcis/mercury/configs/schema_mcis.cfg"
16 PARAMETERS=$PREFIX"/C/bricks/mcis/mercury/configs/params_mcis.conf"
17 VERBOSITY="20"
18 LOG_DIR="/home/hansvh/dmms/logs"
19 BOOTSTRAP="mcis_bootstrap_dmms"
20 DEBUGLEVEL="ANA_ALL"
21 #-----CHANGE VARIABLES ABOVE----------------------------------------------------------
22
23 NODEID=$1
24 CONTROL_GATE=`ls --sort=time /tmp/anaControl_gatesPlug_* | head -n 1`
25
26 echo "Starting MCIS Peer brick..."
27
28 sudo LD_LIBRARY_PATH=$LD_LIBRARY_PATH:$PREFIX/lib/:$PREFIX/C/bricks/mcis/mercury/
       $PREFIX/bin/mcis_peer -n unix://$CONTROL_GATE -a $NODEID -a "-B" -a
     $BOOTSTRAP -a "-H" -a $NODEID -a "--log-dir" -a $LOG_DIR -a "--merctrans" -a
     $TRANSPORT -a "--mercctrl" -a $TRANSPORT -a "--rconfig" -a $PARAMETERS -a "-v
     " -a $VERBOSITY -a "--sampling" -a "1" -a "--selfhistos" -a "1" -a "--
     measurement" -a "1" -D $DEBUGLEVEL
```

Listing 14: Start MCIS Peer brick

D.4 start_mcis_benchmarker.sh

```
1  #!/bin/bash
2  # Hans Vatne Hansen (hansvh@ifi.uio.no)
3
4  #-----CHANGE VARIABLES BELOW----------------------------------------------------------------
5  PREFIX="/home/hansvh/dmms/ana/ana-core/devel"
6  SCHEMA="42;srcip:0:4294967295:true,srcport:0:65536:true,dstip:0:4294967295:true,
       dstport:0:65536:true,bytes:0:1600000000:true"
7  FLOW_FILE="caida_flows.dat"
8  DATA_ELEMENTS="2000"
9  QUERY="1:52830:52839"
10 QUERIES_PER_MINUTE="250"
11 MCIS_IP="129.240.67.93"
12 VERBOSITY="ANA_DEBUG"
13 #-----CHANGE VARIABLES ABOVE----------------------------------------------------------------
14
15 CONTROL_GATE=`ls --sort=time /tmp/anaControl_gatesPlug_* | head -n 1`
16 MY_IP=`/sbin/ifconfig | grep -m 1 "inet addr:" | awk -F: '{ print $2 }' | awk -F\
       '{ print $1 }'`
17
18 echo "Starting MCIS Benchmarker..."
19 sudo LD_LIBRARY_PATH=$PREFIX/lib/:$PREFIX/C/bricks/mcis/mercury/ $PREFIX/bin/
       mcis_benchmarker -n unix://$CONTROL_GATE -D $VERBOSITY -a $MCIS_IP -a $MY_IP
       -a $SCHEMA -a $FLOW_FILE -a $DATA_ELEMENTS -a $QUERY -a $QUERIES_PER_MINUTE
```

Listing 15: Start MCIS Benchmarker brick

D.5 start_system_monitor.sh

```
1  #!/bin/bash
2  # Hans Vatne Hansen (hansvh@ifi.uio.no)
3
4  #-----CHANGE VARIABLES BELOW----------------------------------------------------------
5  ANA_BASE_DIR="/home/hansvh/dmms/ana/ana-core/devel"
6  DEBUG_LEVEL="ANA_ALL"
7  #-----CHANGE VARIABLES ABOVE----------------------------------------------------------
8
9  CONTROL_GATE=`ls --sort=time /tmp/anaControl_gatesPlug_* | head -n 1`
10
11 echo "Starting System monitor...";
12 sudo LD_LIBRARY_PATH=$ANA_BASE_DIR/lib/:$ANA_BASE_DIR/C/bricks/mcis/mercury/
       $ANA_BASE_DIR/bin/node_monitor -n unix://$CONTROL_GATE -D $DEBUG_LEVEL
```

Listing 16: Start System monitor brick

D.6 start_decision_maker.sh

```
1  #!/bin/bash
2  # Hans Vatne Hansen (hansvh@ifi.uio.no)
3
4  #-----CHANGE VARIABLES BELOW----------------------------------------------------------
5  ANA_BASE_DIR="/home/hansvh/dmms/ana/ana-core/devel"
6  DEBUG_LEVEL="ANA_ALL"
7  CPU_UPPER="100"
8  CPU_LOWER="1"
9  RAM_UPPER="999"
10 RAM_LOWER="1"
11 #-----CHANGE VARIABLES ABOVE----------------------------------------------------------
12
13 CONTROL_GATE=`ls --sort=time /tmp/anaControl_gatesPlug_* | head -n 1`
14 PID=$(ps aux | grep "bin/mcis_peer" | head -n 1 | gawk '{print $2;}')
15
16 echo "Starting Decision maker...";
17 sudo LD_LIBRARY_PATH=$ANA_BASE_DIR/lib/:$ANA_BASE_DIR/C/bricks/mcis/mercury/
       $ANA_BASE_DIR/bin/decision_maker -n unix://$CONTROL_GATE -D ANA_DEBUG -a $PID
        -a $CPU_LOWER -a $CPU_UPPER -a $RAM_LOWER -a $RAM_UPPER
```

Listing 17: Start Decision maker brick

D.7 plot_qpm_vs_success.sh

```
#!/bin/bash
# Hans Vatne Hansen (hansvh@ifi.uio.no)
# ------------------------------------------------------------------------
GNUPLOT="/local/bin/gnuplot"
RESOLUTION="700, 400"
LOG_DIR="../logs/"
OUTPUT_DIR="../illustrations/"
OUTPUT_FILE="qpm_vs_success.png"
# ------------------------------------------------------------------------
echo "Plotting for experiment with 1000 data elements..."
$GNUPLOT << EOF
set terminal png size $RESOLUTION
set output '$OUTPUT_DIR/1000_$OUTPUT_FILE'
set title "1000 INSERTED DATA ELEMENTS"
set xlabel "QUERIES PER MINUTE"
set ylabel "SUCCESSFULL QUERY RESPONSE PERCENTAGE"
set xrange[91:309]
set yrange [0:109]
set grid

plot \
  "$LOG_DIR/1000_ra_disabled.dat" using 1:2 \
  title 'RESOURCE ADAPTATION DISABLED' with linespoints lw 2, \
  "$LOG_DIR/1000_ra_enabled.dat" using 1:2 \
  title 'RESOURCE ADAPTATION ENABLED' with linespoints lw 2
EOF

echo "Plotting for experiment with 1500 data elements..."
$GNUPLOT << EOF
set terminal png size $RESOLUTION
set output '$OUTPUT_DIR/1500_$OUTPUT_FILE'
set title "1500 INSERTED DATA ELEMENTS"
set xlabel "QUERIES PER MINUTE"
set ylabel "SUCCESSFULL QUERY RESPONSE PERCENTAGE"
set xrange[91:309]
set yrange [0:109]
set grid

plot \
  "$LOG_DIR/1500_ra_disabled.dat" using 1:2 \
  title 'RESOURCE ADAPTATION DISABLED' with linespoints lw 2, \
  "$LOG_DIR/1500_ra_enabled.dat" using 1:2 \
  title 'RESOURCE ADAPTATION ENABLED' with linespoints lw 2
EOF

echo "Plotting for experiment with 2000 data elements..."
$GNUPLOT << EOF
set terminal png size $RESOLUTION
set output '$OUTPUT_DIR/2000_$OUTPUT_FILE'
set title "2000 INSERTED DATA ELEMENTS"
set xlabel "QUERIES PER MINUTE"
set ylabel "SUCCESSFULL QUERY RESPONSE PERCENTAGE"
set xrange[91:309]
set yrange [0:109]
set grid

plot \
  "$LOG_DIR/2000_ra_disabled.dat" using 1:2 \
  title 'RESOURCE ADAPTATION DISABLED' with linespoints lw 2, \
  "$LOG_DIR/2000_ra_enabled.dat" using 1:2 \
```

```
61      title 'RESOURCE ADAPTATION ENABLED' with linespoints lw 2
62  EOF
63
64
65  echo "Displaying..."
66  xv $OUTPUT_DIR/1000_$OUTPUT_FILE &
67  xv $OUTPUT_DIR/1500_$OUTPUT_FILE &
68  xv $OUTPUT_DIR/2000_$OUTPUT_FILE &
```

Listing 18: Plot queries per minute and corresponding query success percentages

E DVD

This section gives an overview of the contents of the DVD supplementing this thesis. The DVD contains two directories at the root level.

- **ana**: A snapshot of the current ANA development framework and its corresponding bricks.
- **logs**: A complete collection of all the log files from the experiments.

E.1 The ana directory

The ana directory contains the source code for all the current bricks and utilities in the ANA project. It is possible to copy the directory to a writable medium and compile it using the associated config.txt and Makefile. The configuration is currently set up to compile everything needed to repeat our experiments, but changes can be made to include additional bricks. The following are descriptions for the most important sub directories.

C/bricks/mcis/ contains source code for the two bricks forming the MCIS FB, *mcis_peer.cpp* and *mcis_bootstrap.cpp*.

C/bricks/mcis/mercury/ contains source code for the Mercury sub-system utilized by MCIS. This is our version of Mercury, modified for use with ANA. A presentation of this directory and the source code within is given in section 6.3.

C/bricks/resource_adaptation contains source code for the System monitor, Decision maker and MCIS Benchmarker, the bricks related to the resource adaptation.

C/bricks/API/ containts the ANA API implementation shown in Listing 1 and *bin/* contains all the executable binaries after a successful compilation. The remaining directories are of less importance, but contain items like shared libraries and documentation.

The shell scripts from section D are located in their corresponding bricks directories.

E.2 The log directory

All the log files generated from the experiments are located in the log directory. They are separated into directories corresponding to our local and distributed tests. The deeper directory hierarchy contain the different experiment parameter values and a separation between the two MCIS nodes and the MCIS Benchmarker.

The individual log files generated by the ANA log function follow a naming convention like `pid_brickname` in such a way that an MCIS peer log might be named `15228_mcis_peer`.