Experiences with Application Development for Autonomic Networks

Karl-André Skevik, Matti Siekkinen, Vera Goebel, and Thomas Plagemann

Department of Informatics, University of Oslo
P.O. Box 1080 Blindern, N-0316 Oslo, Norway
{karlas,siekkine,goebel,plageman}@ifi.uio.no

Abstract. ANA is a project that examines legacy-free future networking architectures, with a focus on autonomicity. The programming model used in ANA dispenses with the rigid layers of the OSI model and instead uses *bricks* that can be combined to build a *compartment* offering the functionality required by an application. Restrictions such as TCP always being layered on top of IP do not exist, with e.g., arbitrary bricks offering transport functionality being usable to communicate with other nodes in a compartment. Application functionality is divided among specialized bricks, giving a clean and non-monolithic design. We have designed a P2P-like distributed streaming system from scratch, and designed an information sharing system by adapting an existing structured P2P system for ANA. In this paper, we report our experiences on the benefits and pitfalls of application and service development for ANA, and draw some conclusions on suitable design approaches for such novel "disruptive" network architectures.

1 Introduction

Research work on autonomic computing systems has originally been motivated by the effort and complexity of configuration, management, and maintenance of the continuously increasing number of networked computing systems that exist today. The advantage of autonomic networks is obvious in the area of network management because they minimize manual intervention. Making networks autonomic by introducing self-* properties is one of the key elements in many of the recent efforts towards the future Internet, like ANA [1], BIONETS, and CASCADAS[1]. While the challenges of autonomic network solutions receive a strong attention in the research community, little effort has so far been put into the investigation of distributed applications using autonomic networks. Even if self-* properties are introduced into the network, these properties themselves should not be the ultimate goal; instead, the added value of autonomic networks should ultimately be benefits provided to end-users, applications, and application developers.

In order to understand the challenges and benefits that application developers are confronted with when implementing applications for autonomic networks, we

[1] www.ana-project.org, www.bionets.eu, www.cascadas-project.org

analyze in this paper two complementary cases of application development in the ANA Project (Autonomic Networking Architecture). The ANA Project is pursuing disruptive research towards solutions for the future Internet. Besides the development of networking concepts with self-* properties, ANA has introduced an abstract notion of communication starting points to enable the construction of ANA networks without limitations on addressing mechanisms. Furthermore, ANA uses the concepts of compartments composed of smaller bricks to build systems that offer services or application functionality. Each brick offers a simple service and can be used in multiple compartments.

Since ANA does not have a strict layering, like in the OSI reference model, the boundary between network, overlay, and application is fuzzy. Each of these are in fact represented by a compartment. Even all functional blocks running on a node form a compartment. In this paper, we use the term application for the "higher" layer compartments that offer functionality that would be implemented as overlays or applications in a layered architecture. The common factor for these application compartments is that they might need the services of some basic network compartments, such as an IP compartment, for example.

Based on these fundamental concepts, we have designed and implemented two applications: a Peer-to-Peer (P2P) video-on-demand streaming system and a Multi-Compartment Information Sharing System (MCIS) which is essentially a structured P2P system. The P2P streaming system is based on our earlier research on P2P based streaming [2], but the architecture and code has been redesigned from scratch to make use of ANA concepts in order to benefit from the advantages of autonomic networks. In contrast to this fully redesigned system, with the MCIS implementation we have tried to reuse the open source *Mercury* system [3] as much as possible; the MCIS implementation represents the approach of porting legacy applications and overlays to a new autonomic networking architecture. The contribution of this work is a description and analysis of the experiences and tradeoffs of porting versus redesigning and reimplementing applications for future autonomic networks.

Section 2 introduces relevant ANA concepts and gives a short overview of the fundamental APIs used by ANA developers. Section 3 describes MCIS and the process of adapting it for ANA. Section 4 presents the design of a streaming compartment developed for ANA. Section 5 concludes this article with a summary of our contributions.

2 A Glimpse of the ANA Architecture

The objective of the ANA project is a clean slate design of a network architecture for the future Internet with an autonomic flavor. The ANA approach is disruptive in that it does not build on top of the current Internet. Instead, ANA defines a small set of abstractions that form the basic building blocks of the architecture, and that make it possible to host, federate, and interconnect an arbitrary number of heterogeneous networks. These abstractions are *Information Channel (IC)*, *Information Dispatch Point (IDP)*, *Functional Block (FB)*, and *Compartment*.

Protocols and algorithms are represented as FBs in ANA. These FBs can be composed together in any desired manner to produce a specific packet processing functional chain. Communication in ANA occurs towards the start point (the IDP) of a communication channel (the IC). FBs send messages to IDPs which are dispatch points for the next FB. This mechanism makes it possible to recompose the functional chain that represents a specific IC at run time, which is necessary for the control loop behavior of an autonomic system.

A network is represented as a compartment in ANA. ANA does not mandate anything about the internals of a specific compartment, which is free to define inter-compartment communication parameters such as naming, addressing, and routing. Instead, ANA specifies how these compartments interact by introducing a generic compartment API. This API consists of five basic primitives: `publish()`, `unpublish()`, `resolve()`, `lookup()`, and `send()`. The implementation of this API represents the external interface of a compartment. The underlying idea is that a service (represented as a FB) is able to publish itself within a compartment. Users of that service (other FBs) are able to resolve a communication channel (an IC) to the service via that compartment, after which they can send data to each other. An analogy to the layered architecture used today would be an IP FB publishing itself to be reachable at a specific IP address within an Ethernet compartment, enabling other IP FBs to resolve (using the published IP address) an IC to that IP FB via the Ethernet compartment. In addition, compartments can look up specific kinds of services based on keywords if it is not exactly known what to resolve.

The programming model used in ANA is based on *bricks*. There are two kinds of bricks: those that offer access to a network protocol compartment, and those that implement operations such as caching or encryption. Application functionality is divided among specialized bricks, giving a clean and non-monolithic design. A more detailed overview of ANA internals can be found at the ANA web site[2].

The abstractions described above form the blueprint for the architecture and are merely the tools that enable autonomic operations. Establishment of self-* properties naturally requires additional development of corresponding services, such as those that provide resilience or functional recomposition, for instance. Furthermore, various support services are needed, among which one of the most crucial is monitoring services; knowing the current state is imperative for autonomic behavior.

3 MCIS: Multi-Compartment Information Sharing System

For an autonomic network, a generic and fully distributed information sharing service is needed to disseminate monitoring information for decision making. The Multi-Compartment Information Sharing System (MCIS) offers this, by providing lookup and storage facilities for any client brick wishing to share data with other

[2] `http://www.ana-project.org/web/meetings/start`

bricks. The system is rich in capabilities and supports range queries over multi-attribute data records, publish/subscribe functionality, and one-time queries.

3.1 Design and Implementation

In order to create an information sharing service for ANA, we have used an already existing open source system called Mercury [3]. Mercury is a structured P2P system where nodes are logically organized on a ring, in a way similar to Chord [4]. The key difference is that Mercury does not hash the keys, which are the actual attribute values. Hence, each node in the ring is responsible for a particular attribute value range. This difference makes it possible to process range queries efficiently. In Mercury the rings are called attribute hubs and there is one hub for each attribute of a data record. For example, if the data records are two-dimensional coordinates with x and y values as attributes, then Mercury will maintain two attribute hubs and each node is responsible for a particular value range in both of the hubs. Data records are replicated and routed to each of the hubs while queries are routed only to the hub of the attribute that is estimated to be most selective in the query predicate (i.e. specifies the smallest range). This

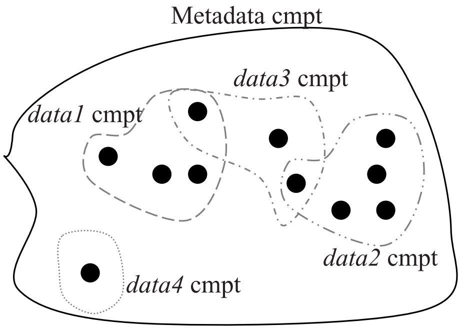

Fig. 1. Data compartments in MCIS

kind of content addressable overlay fits well to the concept of a compartment in ANA. All data records of a specific type naturally forms a *data compartment*. Each data compartment routes messages via its attribute hubs independently of other data compartments. Resolution and publish procedures of data items boil down to query processing and store operations, respectively. There is a special *metadata compartment* which every node belongs to. It contains information about the different data compartments that exist and enables discovery of them (see Figure 1). As the name suggests, the MCIS brick is the entry point to the data compartments and handles all the resolve, lookup, and publish requests from local client bricks. MCIS bricks at different nodes cooperate on the maintenance of a single distributed Mercury system per data compartment.

The Mercury software has been designed in a very modular way. The code for the overlay management and routing is located in the mercury package, while code for the underlay networking layer can be found in packages such as *wan-env* for TCP/UDP communication over IP using the Berkeley sockets API. Establishing point-to-point communication with other nodes in ANA requires being able to resolve the other end point via some compartment, where that end point has previously published itself. Thus, we needed to implement *ana-env* as the underlying ANA networking layer. Ideally this layer would choose, from the available network compartments, one that is most suitable for reaching a particular end point in the current situation. However, in the current implementation the network compartment must be chosen with a parameter when the system is started.

In addition to the networking layer, we needed to build the MCIS brick functionality; the handling of the resolve, lookup, and publish requests as specified by the generic compartment API. The brick functionality can be seen as a kind of adaptation layer, from the generic compartment API to the Mercury specific API and message types.

3.2 Lessons Learned

One particularly cumbersome issue we had in incorporating Mercury into ANA was the use of identifiers. ANA as a "disruptive" future networking architecture does not mandate the use of IP or any other protocol for communication, while the current Internet makes almost exclusive use of IP. As a consequence, as with most software designed for the Internet, Mercury assumes the use of IP addresses and port numbers as identifiers of services and clients. As a consequence, we needed to modify major parts of the entire source code to introduce strings as generic identifiers.

The main reason why it has been quite straightforward to use Mercury as the basis for building the MCIS brick has been the modularity of the source code. While it likely stems from the fact that the authors of the original code wanted to use a simulated networking layer in addition to the sockets API. Apart from the changes needed to support more generic identifiers, this separation between network layers enabled us to use the original code with only minor modifications.

Our work on Mercury has shown how porting legacy Internet networking software to ANA with only a minimum of modifications can be done, but there is still room for further integration. The original software could be decomposed into smaller consistent functionalities, which could then serve in several contexts. For example, the load balancing functionality of Mercury might also be usable by other compartments.

4 Streaming Compartment Design

For real-time applications such as video streaming systems, having accurate information about the network, such as delay or available bandwidth between nodes, can be an important factor in achieving good performance. Unfortunately, this kind of information is not easily obtainable on the current Internet. Several techniques for doing this, such as [5], have been proposed, but while it is possible to keep code for this kind of functionality in an external library rather than inside the application, the underlying limitations do not change: the Internet does not provide an interface for obtaining all the information required by the application, and a library might require updates as the interconnect technologies of the Internet changes.

However, in ANA, monitoring is provided as a fundamental service on all nodes. Furthermore, ANA has been designed to support having accurate information provided by routers and other intermediate nodes. As we have previously designed a P2P video streaming system for the Internet [2], we have been interested in seeing how having the monitoring services available in ANA might affect the design of a similar system.

4.1 Compartment Overview

We have designed a distributed compartment that offers video streaming services, with content retrievable from participating nodes that have previously retrieved the same content. This kind of system requires a mechanism for transmitting media data between nodes, but also a metadata handling system that offers file search functionality and a way of keeping track of the nodes that have copies of the data. A typical usage scenario would be for a user to search for and request a movie, upon which a list of nodes with the content available would be obtained, and the content requested from nodes on the list. The downloaded parts of the movie would simultaneously be made available to other users. Files are divided into blocks to make information about downloaded files easier to share and manage.

Some functionality is clearly the same regardless of whether an application has been designed for the Internet or ANA; disk caching and media playback occurs after the media data has been retrieved from the network and can be identical. What reveals the special characteristics of ANA is how the application is structured. A typical Internet application use have a *poll()* loop, or multiple threads, to multiplex connections to other nodes, but it is common to have the majority of the functionality implemented as part of a single application, perhaps even running as a single process. If there is any code shared between applications, it is primarily in the form of shared libraries that implement common operations.

Rather than building monolithic applications, the ANA application development concept is based on the principle of combining many small *bricks* to form a larger structure, or compartment, that offers application functionality. This might seem similar to the use of libraries, but a brick is an actively running service that can be included in the combined structure of multiple compartments. As with libraries, this gives the benefit of sharing code, but there are additional benefits that come from bricks being a running and shared service. The first is with regard to efficiency, especially in the context of network monitoring. An Internet application that wishes to estimate the transfer speed or available bandwidth to another node can do this by transmitting specially crafted packets [5], but if each application does this independently the result will be redundant traffic that increases the load on the link between the two machines needlessly. However, if all applications request this information through the same brick, the result will be less overhead and shorter response time if the answer is already known. Another important benefit comes from the possibility for the application to adapt to changes; the system can react to changes in the network by changing the bricks that offer various types of functionality. To use the network monitoring example, on the Internet, it might only be possible to use heuristics to estimate link speeds and a monitoring brick for the Internet would use these kinds of techniques, but on a network where ANA nodes provide additional monitoring functionality, the routers can be queried directly. An ANA node can react to these two different scenarios by replacing the relevant monitoring brick. The design of ANA gives applications the inherent possibility of having this kind of adaptability, and this approach can be used in many situations: in order to change transport protocols, insertion of transcoding bricks to reduce media size on low bit rate links, etc.

The use of bricks has a direct influence on the structure of the application; the streaming functionality is not offered by a single brick, but by a set of bricks that work together. One immediate benefit of this is that some of the required functionality is already offered by the system; network monitoring is provided by the monitoring framework. What remains is a way to exchange media data, metadata handling, and the interface to the media player. To demonstrate the flexibility of the brick system, we have designed two variants of the streaming system. The first is server based and similar to a traditional video streaming system suited for commercial streaming, with data originating from a server maintained by a content provider. The server also provides metadata handling services such as file search. The difference from a traditional client/server system is that clients serve downloaded content to other clients; the server is the origin

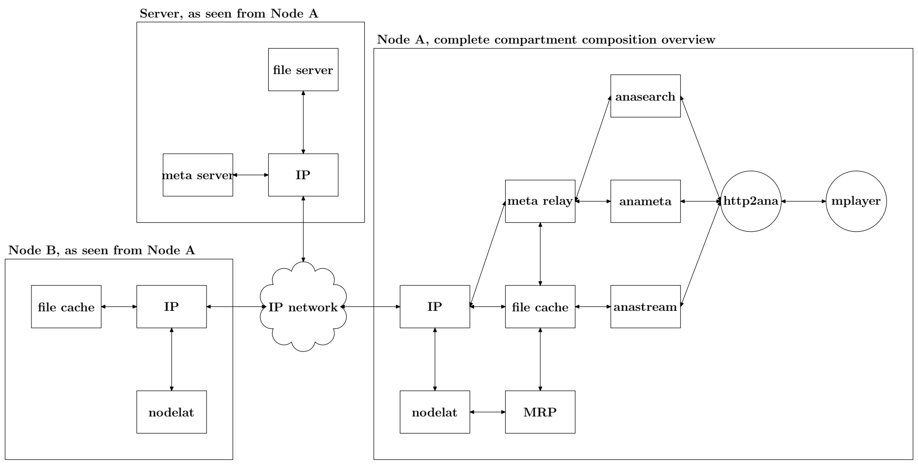

Fig. 2. Server based streaming compartment

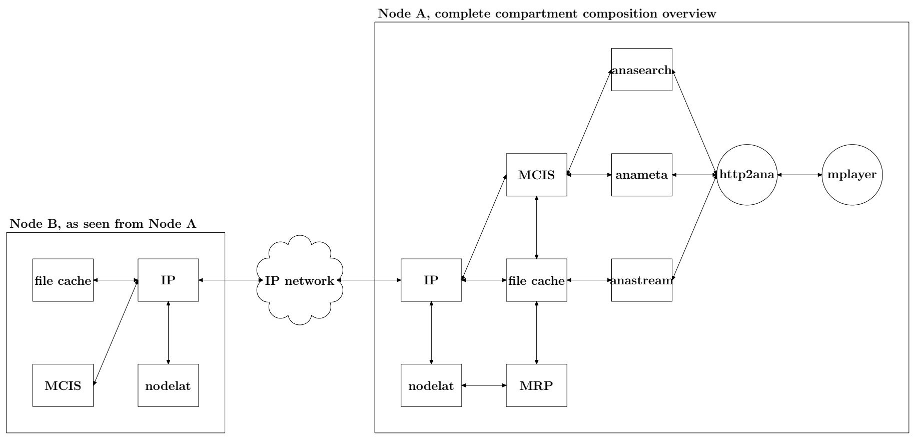

Fig. 3. Distributed streaming compartment

of all content, but not necessarily the only location to retrieve media data from. The structure of this variant is shown in Figure 2.

The second variant is a completely distributed system, with no centralized servers. Any user can publish files, resulting in a system similar to P2P file sharing networks such as eDonkey [6]. The decentralization is achieved through the use of the MCIS brick, which provides metadata handling. The structure of this system is given in Figure 3, and as can be seen, despite being two completely different systems, the only difference in the brick composition of *Node A* is that the brick managing metadata has been replaced; the rest of the system is identical.

4.2 File Server Brick

The server based design initially provides all content from the server maintained by the content provider. The *file server brick* handles byte-range requests from client nodes. Files are identified by a message digest, which is obtained by clients through the metadata brick. An IP brick is used for communication with clients.

4.3 Metadata Server Brick

Metadata in the server based streaming compartment is managed by the *metadata server brick*. For this usage scenario all files are maintained on the server; users can report having downloaded parts of a file, but cannot add new files to the system. Metadata queries only search metadata for the files located on the server.

The design of this brick reflects one of the places where a decision had to be taken with regard to the granularity of bricks. The brick performs tasks relevant for a single type of operation, namely metadata management, but there are two types of metadata: the digest, node, block, 3-tuple, and the information relevant for media files, such as movie title and quality. An alternative approach would have been to have two bricks instead of one, and it is conceivable that either brick would be useful by itself for other compartments. In general, the two opposite extremes for brick design is to have either a small number of complex bricks, or a large number of small atomic bricks. The first results in code complexity, while the second increases system complexity. We have chosen a solution in the middle, with bricks divided based on the conceptual operation they provide, rather than the low-level operation they perform.

4.4 File Cache Brick

The *file cache brick* is an important part of both the server based and distributed system. As the brick composition overview of either system shows, this is the most connected brick, with communication links to four other bricks. The task of this brick is to manage the file cache on end user nodes. This includes managing the file data stored on disk, and retrieving missing data from either the file cache or file server brick on other nodes.

As with the metadata server brick, this brick performs multiple tasks that could have been implemented in separate bricks. For example, answering data requests from remote and local nodes could have been done by two different bricks. The same is the case with the data retrieval functionality. In practice however, it was found that all these operations were quite tightly connected, and having them in separate bricks would have required synchronization mechanisms to ensure safe handling of the data stored on disk and in memory, adding complexity and essentially defeating the purpose of separating the bricks.

4.5 Meta Relay and MCIS

In both the server based and distributed streaming scenarios, the metadata requests pass through a single block. In the server case, this is the *meta relay block*, which simply relays all requests to or from the meta server block on the server node. Because of this, the interface is essentially the same as that given for the meta server brick, except that the brick is not available through the IP compartment, only to the bricks on the same node. When the MCIS is used for metadata handling, the metadata requests are processed in the context of the MCIS compartments. Support for other ways of managing metadata can be added by simply writing a new brick that supports the same interface as these two bricks.

4.6 Application Bricks

The three application bricks, *anasearch*, *anameta*, and *anastream*, have been designed to be used as normal shell commands, and each provide a simple interface to the ANA streaming system. Of these commands, anasearch and anameta are simply wrappers around the meta brick, allowing a user to search for and obtain information about available files. The anastream brick returns a media data stream for a specified file.

Again, also in this case it would have been possible to handle the brick division in a different way. Metadata queries are performed by two bricks; *anastream* and *anameta*, and the functionality of these bricks could arguably have been implemented in a single brick. The reason for the division in this case is that integrating ANA bricks with the UNIX shell is currently somewhat cumbersome, and keeping the two metadata query types in different commands makes it possible to have more easily understandable interfaces for the commands.

4.7 HTTP Proxy Based ANA Gateway

The *http2ana* and *mplayer* elements in Figure 2 and Figure 3 are not ANA bricks but standard UNIX applications. To demonstrate that ANA can be used with a normal media player, we have created a gateway application that functions as a normal HTTP proxy, that, rather than obtaining data from a web server, uses the application bricks to retrieve the data over an ANA network. Any media player that supports streaming over HTTP, and has support for HTTP proxies, should be usable with this gateway.

4.8 Non-compartment Bricks

The core part of the streaming functionality is provided by the file cache brick and the metadata handling bricks, but network monitoring is important for performance reasons, and node monitoring functionality is provided by the MRP brick[7]. The information provided by the MRP brick includes simple status information such as whether a node is available or not, but can include more complex queries such as a request for an ordering of the nodes based on criteria such as latency.

MRP operations manipulate so-called *nodesets*, that can consist of an arbitrary set of nodes. Operations include adding nodes to a nodeset, removing nodes, and requesting orderings of the nodes based on various criteria. The MRP brick does not perform any measurement operations itself, it merely manages a set of nodes, and sends requests for network measurements to the monitoring framework, here represented by the *nodelat brick*. This brick measures the RTT between the node itself and a different node. Each nodelat brick currently exposes the latency measurement interface to other bricks on the same node, but it would be possible to expose it to other nodes, allowing these nodes to measure the RTT between arbitrary sets of nodes.

The final brick which is part of the streaming compartment is the IP brick, which provides IP transport functionality.

4.9 Lessons Learned and Limitations

The current status of the streaming compartment is that all the bricks needed for the server based scenario have been implemented and are currently undergoing testing. During the development process we have made several observations. We have especially found that the brick concept leads itself well to easy development and code testing. Most bricks are fairly small and provide a single operation through a well-defined interface. The brick based application construction encourages having small bricks that are simple and consequently, ideally easy to understand and test. Compared to the development of the P2P video streaming system we have previously created for the Internet, it has been much simpler to develop and test multiple separate bricks than one large application. It should be noted that the Internet application was more feature rich, but changing the brick based design is much simpler, as can be seen in the ease with which the structure of the system is changed from being server based to being fully distributed, by simply using a different brick for metadata handling. Furthermore, having a monitoring framework has made it possible to add network awareness without having to implement code for this in the application.

While the underlying principles of ANA provide several benefits for application design, there are some practical issues that have affected brick development. The ANA project looks at legacy-free networking design, and a consequence of this is that functionality that is taken for granted on the Internet needs to be reimplemented from scratch. As the ANA code base is still far from mature, there are some limitations that have influenced the design. One obvious oddity is the use of IP to transport media data, and the system does in fact do streaming directly on top of IP packets. The reason for this is that there are currently no higher level protocols implemented; only Ethernet and IP transport without packet fragmentation. The consequence is that as opposed to having a simple stream-like interface, the bricks need to consider factors such as packet size .All requests and replies are currently kept below the MTU, which complicates request handling. Lack of a reliable transport protocol makes it necessary to handle retransmission of lost packets in the application. However, these limitations should disappear as more advanced bricks become available. It could rather be argued that these limitations demonstrate the flexibility of ANA, as it is possible to implement a distributed streaming system on top of a simple IP implementation. The functionality offered by the MRP brick is similarly limited due to the lack of network measurement bricks.

5 Conclusion

In this article, we have described work on application development in the context of ANA. A central ANA concept is the separation of functionality into bricks that offer a single type of service, and the combination of bricks to form compartments that offer more complex functionality. We have shown how this approach affects both the porting of the MCIS information sharing system to ANA, and the development from scratch of a distributed video streaming system. There are still limitations that complicate application development, but these issues are a result of functionality that has still not been implemented rather than limitations imposed by the design of ANA. More than being a deficiency, being able to do streaming directly over IP demonstrates the flexibility of ANA. Our experience shows that the brick concept is well suited for application development.

Furthermore, we have shown that the ANA compartment concept fits with currently used networking paradigms, using a content addressable network in the form of data compartments. Importing legacy software can be straightforward but usually requires a kind of adaptation layer in order to conform to the generic compartment API. A complicating factor can be the implicit assumption of IP addresses as locators and identifiers in existing application design. Such design decision often influence the entire source code and, therefore, require modifications throughout the code of a legacy application when imported to an identifier/locator agnostic environment.

There are many similarities between the brick concept and the use of shell commands in UNIX. If bricks could be combined in an easy way similar to shell commands, with a language designed for this purpose, it would make the brick concept even more powerful, and simplify the creation of complex networking applications. Especially important in this context is the existence of a monitoring framework, which can be assumed to exist on any ANA node, and which allows easy integration of monitoring operations into a compartment.

As for future work, it consists of two directions of research. First, to complete testing and integration of the compartment. Second, to further examine the applicability of the brick concept to application development, by trying to identify basic functionality that is common to many applications, and how these can be interfaced via a high-level shell-like language.

Acknowledgment. This work has been funded by the EC-funded ANA Project (FP6-IST-27489), and supported by the CONTENT Network-of-Excellence.

References

1. Jelger, C., Tschudin, C.F., Schmid, S., Leduc, G.: Basic abstractions for an autonomic network architecture. In: WoWMoM 2007: Proceedings of the 2007 International Symposium on a World of Wireless, Mobile and Multimedia Networks (2007)
2. Skevik, K.A.: The SPP architecture – A system for interactive Video-on-Demand streaming. PhD thesis, University of Oslo (April 2007)
3. Bharambe, A.R., Agrawal, M., Seshan, S.: Mercury: supporting scalable multi-attribute range queries. In: SIGCOMM 2004: Proceedings of the 2004 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 353–366. ACM Press, New York (2004)
4. Stoica, I., Morris, R., Karger, D., Kaashoek, M.F., Balakrishnan, H.: Chord: A scalable peer-to-peer lookup service for internet applications. In: SIGCOMM 2001: Proceedings of the 2001 conference on Applications, technologies, architectures, and protocols for computer communications, pp. 149–160. ACM Press, New York (2001)

5. Carter, R.L., Crovella, M.E.: Server selection using dynamic path characterization in wide-area networks. In: INFOCOM 1997: Proceedings of the Sixteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Driving the Information Revolution, 1014 (1997)
6. Heckmann, O., Bock, A.: The edonkey 2000 protocol. Technical Report KOM Technical Report 08/2002, Darmstadt University of Technology (2002)
7. Skevik, K.A., Goebel, V., Plagemann, T.: Design, prototype and evaluation of a network monitoring library. In: Rong, C., Jaatun, M.G., Sandnes, F.E., Yang, L.T., Ma, J. (eds.) ATC 2008. LNCS, vol. 5060. Springer, Heidelberg (to appear, 2008)