Virtual Overlays: An Approach to the Management of Competing or Collaborating Overlay Structures

Paul M.Okanda[1], Sebastian Steinhauer[2], Gordon Blair[1]
[1]Next Generation Middleware Group, Computing Department,
InfoLab21, Lancaster University, Lancaster, LA1 4WA, UK
{okanda, gordon}@comp.lancs.ac.uk

[2]Business User Imagineering, SAP Labs, LLC,
Palo Alto, CA, USA
sebastian.steinhauer@sap.com

Abstract. Overlay networks are a technique whereby application developers create virtual customized networks on top of physical networks. Recent implementations of peer-to-peer applications such as file sharing and VoIP have increasingly meant that overlay networks have almost become ubiquitous. As a result, future overlay networks will increasingly coexist on the same node. A number of middleware frameworks such as GRIDKIT [1], P2 [2] and ODIN-S [3] currently offer support for the co-existence of multiple overlay networks. However, co-existing overlay networks interfere with each other's performance either through competition for resources or the lack of collaboration between them. This paper introduces an approach called virtual overlays which manages competition and collaboration between co-existing overlay networks in a way that is expressive, flexible, configurable and dynamically adaptable.

Keywords: Overlay Network, Virtual Overlay, Middleware.

1 Introduction

An overlay network can be seen as an application level network layer or partial network stack which represents a virtual network. This virtual network is realized as a composition of nodes and logical links abstracting from an underlying existing network. The main motivation behind overlay networks is the provision of more tailored application services to support applications in different domains e.g. multimedia file sharing, peer-to-peer networks etc. Overlay networks have gained widespread utilization in recent years as a way through which services offered by the underlying physical network can be tailored to better support the requirements of applications. Applications such as multimedia file sharing and Virtual Private Networks (VPNs) have proven that overlay networks provide a powerful and efficient solution for specific problems, e.g. security and content distribution.

The success of current overlay networks e.g. Chord [5], SCRIBE [6] and Pastry [7] has meant that future trends will result in nodes that run different distributed applications hosting multiple overlays at a time. This is bound to introduce competition for local resources such as CPU time, memory consumption and network resources such as bandwidth. There is therefore a need for a framework that resolves

competition for local and network resources, manages collaboration between two or more overlay networks and, creates a higher level of abstraction that provides developers with better control over the management of resource conflicts and collaboration between overlay networks.
We propose the use of *virtual overlays* as a means through which the strengths of multiple overlapping overlay networks can be combined to not only efficiently resolve conflicts between overlay networks but also manage competition between them and support their collaboration in a flexible, adaptive and configurable way.

2 Background on Overlays

2.1 Definition of Network Overlays

An overlay network can be defined as an application level network layer or partial network stack which represents a virtual network. This virtual network is realized as a composition of nodes and logical links that are an abstraction from an underlying existing network. The main motivation behind the implementation of overlays is to provide more application-specific or tailored network services which are not provided by the underlying network. The advantages of overlay networks are pointed out in Aberer et al [4] thus:

> "In principle, distributed application services could also use directly the physical networking layer for managing their resources, but using an overlay network has the advantage of supporting application specific identifiers and semantic routing, and offers the possibility to provide additional, generic services for supporting network maintenance, authentication, trust, etc., all of which would be very hard to integrate into and support at the networking layer."

This general idea of overlay networks is well known and has been shown to work well. Transmitting information by sending telegraphs on top of a circuit switched network can be seen as a historic example. Dial-up connections between computers and bulletin board systems are an example that is close to the type of overlay that is the subject of this research paper. Modern overlay networks are a critical part of distributed applications. For instance peer-to-peer applications use overlay technologies to create virtual networks as an abstraction from heterogeneous underlying networks, Virtual Private Networks (VPNs) add authentication and encryption to messages sent through them, which often cannot be provided by the underlying networks while peer-to-peer applications are ubiquitous and are used broadly.
Current software systems that utilize overlay technologies, e.g. Chord [5], SCRIBE [6], or Pastry [7], usually implement a specific well known and well defined overlay routing mechanism and a corresponding topology. These virtual networks act like classic, message based, networks on top of underlying networks. They can be used in a stacked manner, but they keep their basic topology. Hence if an overlay network

layer is designed as a ring network it maintains this structure when stacked with other overlay networks.

2.2 Why Virtual Overlays?

Current overlay networks have been shown to provide software engineers with high levels of abstraction at the cost of fine grained control on the message transmission. The wide adoption of overlay technologies results in co-existing implementations executing in nodes, e.g. the deployment of a VPN and a peer-to-peer file sharing application on a single computer. The new concept presented in this paper aims to provide an approach for controlling and orchestrating multiple coexistent overlay networks. In order to address the requirement mentioned in the introduction, virtual overlays provide a technology that can be used to a) resolve resource conflicts, e.g. competition for memory between an application specific implementation of a multicast overlay network and an unreliable transmission overlay network, b) manage collaboration between coexisting overlay networks e.g. between an overlay providing reliable transmission and an overlay providing multicast without forcing developers to give up the advantages of application-specific overlays and, c) provide a higher-level abstraction that gives developers the ability to configure the behaviour of overlays at a fine grained level, i.e. on a per message basis. This fine-grained per-message manipulation of behaviour implies that an overlay's behaviour is not only dependent on its static forwarding mechanism but also on the message which is passed. This also implies that technologies which are usually used on a packet level within the ISO/OSI layers can be used to orchestrate overlay networks.
As will be seen in the next section, the focus of the design and implementation of the proof-of-concept system is on the manipulation of message routing in overlay networks at a finer level of granularity and in a more flexible way, without having a significant negative impact on understandability or system performance.

3 Design and Implementation of the Virtual Overlay

In this section, we describe our design of a virtual overlay by presenting a background on GRIDKIT and demonstrating how the design is realized on top of the GRIDKIT middleware framework. Crucially, the goal is to provide an approach that manages competition and collaboration between multiple overlay structures.

3.1 Background on GRIDKIT

GRIDKIT is a middleware solution whose aim is to provide support for the development of complex distributed systems. It can be used to develop a range of approaches some of which are service-oriented. In order to provide an array of interaction types, GRIDKIT provides different plug-able overlays at different levels of abstraction. The set of services provided by the GRIDKIT middleware for grid environments consists of service bindings, resource discovery, resource management and security. All of these can be combined with the communication layer realized by

the GRIDKIT Overlay Framework [1]. GRIDKIT addresses the common challenges of middleware systems by providing developers with the possibility to interconnect overlays in different ways. In their paper 'GRIDKIT: Pluggable Overlay Networks for Grid Computing', Grace et al [1] summarize the main goal of the GRIDKIT framework as follows:

> "The goal of our research in this area is to develop ways of building fully customizable, extensible, and evolvable overlays by factoring out generic techniques and protocols (e.g., large-scale neighbor discovery, and network capability discovery techniques), and enabling these to be composed, extended and dynamically reconfigured under the auspices of a well - defined [component frameworks]."

As described above, overlay networks can be used to create functionality on top of underlying networks. In GRIDKIT, the developer is given the freedom to combine different overlay networks or even components of different overlay networks to create custom overlay networks. These custom overlay networks are created by interconnecting OpenCOMJ [1] components and component frameworks (CFs). This provides software developers with the tools to create custom interaction types based on pre-existing building blocks. This approach gives software architects more flexibility during the design of their applications [9], [1]. As described above, every overlay consists of OpenCOMJ components [1]. As shown in **Figure 1** below, an overlay has to control its topology and provide its forwarding technique. Since the topology management and the forwarding are based on shared information, a third component is used to provide state information.

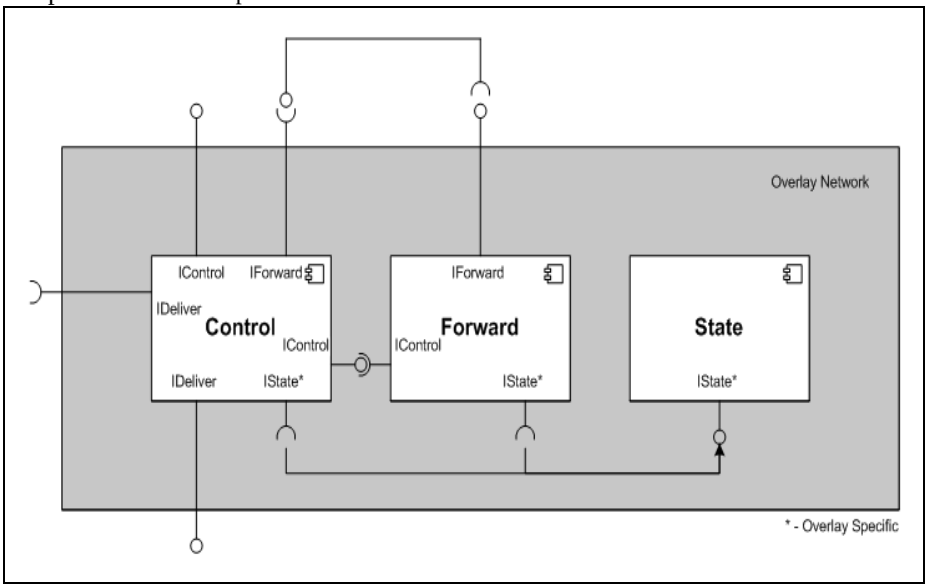

Figure 1 An Architecture for a GRIDKIT Overlay

Each overlay layer needs to be connected to an `IDeliver` interface and provides an `IDeliver` interface. This interface is used in the GRIDKIT framework to pass messages between layers of stacked overlays. An integer value is used to identify the overlay class that a message is passed to. The `IDeliver` interface is used to pass

messages upwards through the overlay stack. Each overlay also provides and consumes at least one `IForward` interface. In contrast to `IDeliver`, the `IForward` interface is used to pass messages downwards through the overlay stack. High level control functionality is accessed using the `IControl` interface as it can be used to join and leave overlay networks.

3.2 Extensions on GRIDKIT to support Virtual Overlays

The system is implemented as part of GRIDKIT which features support for the co-existence of overlay networks.

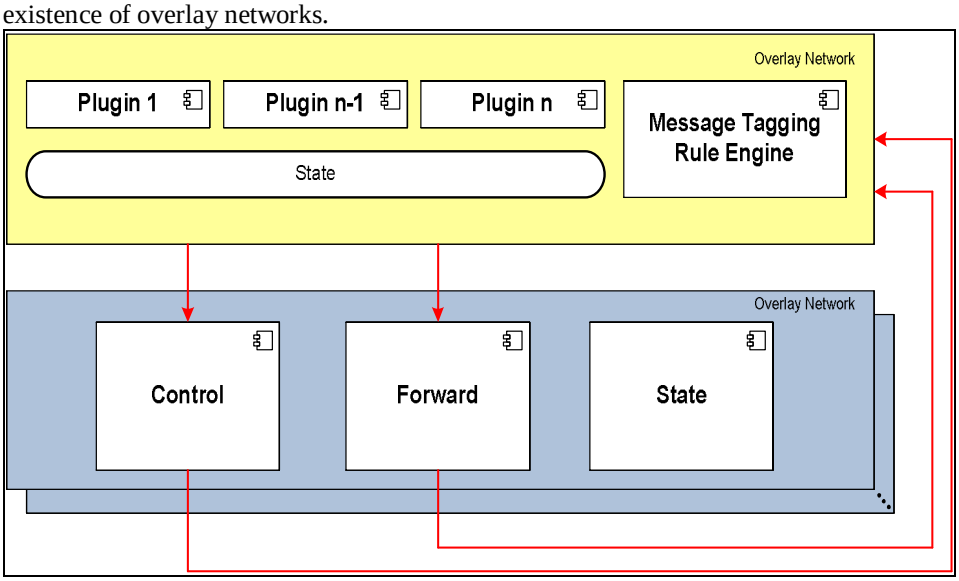

Figure 2 An Architecture for a Virtual Overlay built on an Overlay

As shown in **Figure 2** above, a virtual overlay component intercepts all messages sent, received or forwarded by native overlay networks. Depending on the content of the messages and the functions implemented by the plugins and the rule set deployed by the virtual overlay, these intercepted messages may or may not be re-injected into the native overlay networks. Note that this effectively equates to a meta-level approach to the management of overlays (meta-overlays) and indeed this is implemented using GRIDKIT's underlying reflective mechanisms. Below, we describe the fundamental constituents of the virtual overlay.

Message Tagging & Rule Engine - The general concept of tagging messages was inspired by IP Filters [12] and Conoboy et al's [11] work on the rule language and its influence on packets being processed by the packet filter. All messages are tagged by all applicable rules; the process of tagging does not alter the message itself but adds a flag to the message for each applied rule. After adding all applicable flags to the message, a set of filters is used to alter the message according to the flags the message was tagged with. To ensure consistency with the idea of an interchangeable rule engines, the central requirement for a rule engine in the context of this project is its full compliance to JSR 94 [13]. This specification defines the general interfaces a rule

engine needs to implement without specifying a rule definition language or a specific technology for the rule engine. Since the main differences for rule engines in this context are the technology and the language used to define the rules, the number of candidates for the virtual overlay's implementation was fairly limited. The following criteria were used during the decision making process: complexity, license, rule language, community, and documentation. Amongst the rule engines evaluated were Jess [14], JBoss Rules [15] and Hammurapi Rules [16]. Although the Hammurapi Rules development community is relatively small, its lightweight implementation made it the best choice for a prototype system.

Plugins - The intercepted messages are tagged according to a rule set and based on the tagging of each message, control plugins manipulate the message. Finally the message is injected into or sent via a native overlay. Each plugin checks whether a message contains specific tags and if the message does, it performs some action, otherwise the plugin executes its default action.
Crucially, the virtual overlay does not only intercept messages within the existing overlays but is in itself an overlay which can be stacked on top of existing overlays. It can receive messages from other overlays and send messages using the default set of overlay interfaces. In order to offer support for the orchestration of overlays, the system's overlay components implement an interception and injection interface.
Figure 3 below illustrates a plugin that checks whether a message contains specific tags. If the message does, it performs some action, otherwise the plugin executes its default action.

```
public class DropPlugin
        extends GenericPlugin
        implements IPlugin, IConnections, ILifeCycle, IUnknown, IMetaInterface{

        public DropPlugin(IUnknown runtime) {
                super(runtime);
        }

        @Override
        public MessageDecorator processMessage(MessageDecorator message) {
                if (    (message!=null)  &&
                                (message.Tags.contains(DROP.getInstance())) &&
                                (!(message.Tags.contains(PASS.getInstance())))

                        ){
                        message = null;
                }

                return super.processMessage(message);
        }

}
```

Figure 3 Java Source Code for a Sample Filter Plugin

The method `processMessage` checks if a message was passed to it, and it was it checks for the tag `DROP` and the absence of the tag `PASS`. If a `DROP` tag is found and not a PASS tag the message is set to null. If a `PASS` tag is found or no `DROP` tag is found the message is not manipulated. In either case, the message or null is passed to the next plugin by calling the parent method `processMessage`. This ensures that the entire chain of plugins is processed and the default action is carried out. Cases which require to process all messages can be imagined hence it is required that the messages are passed down the chain even if they are null. The inherited class `GenericPlugin` also implements the entire OpenCOMJ functionality. The

development of a plugin only requires overloading the `processMessage` function - if a behavior different from just passing the message is required.
The tagging engine in this prototype was developed as a façade around the rule engine to tag messages. In order to show that the tagging of messages with a following processing of the messages based on their tags is an efficient way of manipulating messages on middleware level a small rule set was defined. **Figure 4** below shows how a rule can be created by implementing the infer method in a class inheriting from Rule.

```
public class TagDROP extends Rule {

      public void infer(MessageDecorator msg){
            String PATTERN = Arrays.toString("DEBUG".getBytes());

            if (Arrays.toString(msg.RawData).indexOf(PATTERN)>-1){
                  msg.Tags.add(DROP.getInstance());
            }
      }
}
```

Figure 4 A sample rule for Hammurapi Rules

The next section details a set of experiments that were developed over the GRIDKIT overlay framework.

4 Experimental Evaluation

This section details an experimental evaluation of the implementation of the design discussed in **section 3.2** above. To facilitate the experiments, two representative and existing overlay networks are used to prove the concept of the described system; Tree Building Control Protocol (TBCP) [8] and Chord [5]. TBCP is used to span a balanced application level multicast tree. While Chord represents a distributed hash table (DHT)-based overlay ring network. Chord is a well known overlay network while TBCP is a clean realization of an application level multicast tree. Implementations of both overlay networks are part of the GRIDKIT framework.
From our implementation of the design detailed in **section 3** above, we set up two sets of incremental experiments that focused on validating the architecture described in the previous section. As a first step, the general concept was verified by implementing a sample application that could be used to show major aspects of the proposed system and its basic performance metrics. Since overlay networks are not only defined by the forwarding technique that they implement but also by their topology and their state, the components maintaining and realizing their topology and state are represented in the context of virtual overlay networks. The second set of experiments aims to show a non-static (dynamic) implementation of meta-routing. It presents a prototype developed for inter-overlay routing based on self-configuring routes.

4.1 A Basic Middleware Firewall

Overview
The proof-of-concept implementation presented in this sub-section shows the realization of interception of messages within an overlay and reinjection of messages in the very same overlay. Crucially, its aim is to a) illustrate the internals of a minimal configuration of a virtual overlay and b) evaluate the performance overhead that is introduced by the implementation of a virtual overlay.

Implementation
As detailed below, this experiment implements the three constituents of a virtual overlay described in section 3 above. It involves a selection and implementation of a message injection mechanism, an implementation of a message tagging technology and an implementation of a rule engine.

Message Injection – The fundamental concept of the proposed architecture is message interception and message injection. To prove this concept, a test application comprising two parts, a sender and a receiver was developed. Both components intercept messages before sending or receiving them. The intercepted messages get manipulated and then re-injected into (other or the same) overlay networks. In the initial stages of the experiment, a TBCP tree containing exactly two nodes was created, one node being the sender while the second node acted as a receiver. The sample application used a custom Log4J[10] appender to published messages to a multicast tree. Since broadcast messages were filtered within the sender and receiver in a multicast application that was extended to provide support for the interception of inbound and outbound messages, this could be considered a simplified firewall.

Message Tagging - In the first prototype implementation, the filter basically drops or passes messages according to their tags. To gain higher flexibility, the plugin processing the tags has to define a default behavior in case no matching tags can be found or in case conflicting tags are attached to a message. The mechanism of using a separate tagging engine which does not define the behavior of the stack creates flexibility in choosing a rule engine for a particular task, or to meet specific environmental constraints. It also allows the use of precompiled sets of tags to be attached to precompiled messages, which might be relevant in throughput-critical systems.

Rule Engine - As illustrated in **Figure 5** below, the TBCP Overlay was extended to provide the interface `IOverlayCallback`. It also provides intercepted messages to the Virtual Overlay component using the `IIntercept` interface. The Virtual Overlay component uses a Tagging Engine component to wrap the rule engine via the interface `ITag` and forwards tagged messages to a chain of plugins using the `IPlugin` interface. The diagram below shows the components used. The chain of plugins only consists of two generic empty plugins as proof of the concept of the plugin chain as well as the `DROPPlugin`. The chain of plugins is realized using the `IPlugin` interface that each plugin has to implement.

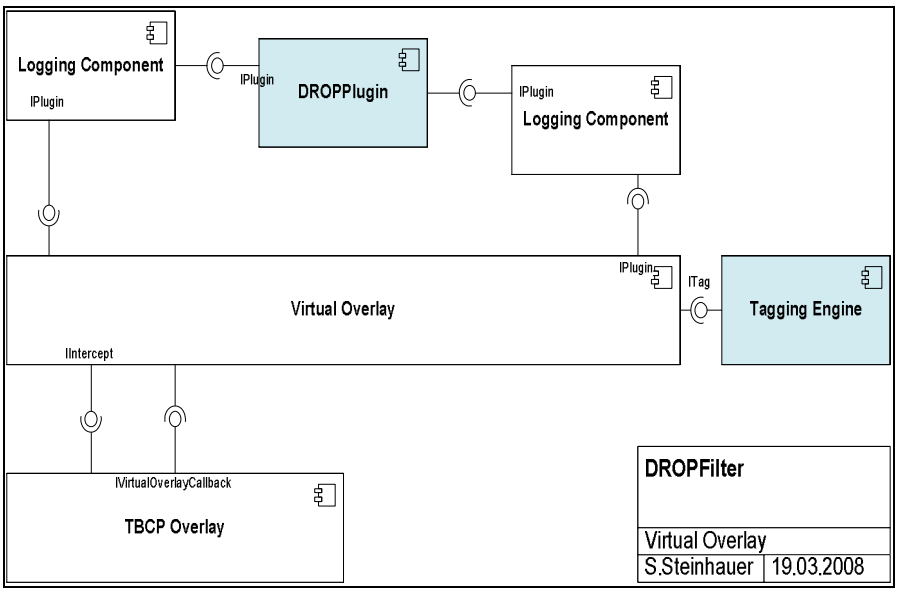

Figure 5 Major components of the first prototype

It is also implemented by the Virtual Overlay component which re-injects messages into the TBCP Overlay after they have completed the entire chain. The prototype's major components are briefly described below.

The Tagging Engine – This rule tags all messages containing the word "DEBUG" with the tag DROP. A similar rule is used to tag all message having the word "FATAL" in them with the tag PASS. All other messages are not tagged at all. The rule engine automatically loads rules listed in a rule set definition file. The Tagging Engine also checks the rule set definition file for updates before tagging a message. This very simple approach allows runtime manipulation of the deployed rule set and was implemented to show that runtime adaptability can be achieved using the proposed system.

Evaluation

To evaluate the prototype, a basic system creating log messages was used. The generator sends two sets of 1000 numbered messages with priorities iterating over {DEBUG, INFO, WARN, ERROR, FATAL}. A small receiver application logged the message, including the time of creation, as well as the time of reception to a file. To provide reproducible and facilitate comparison of results without the need to consider time synchronization, the receiving and sending application executed on the same computer. All involved Java Virtual Machines (JVMs) were running with normal priorities as user applications. Two virtual machines were used to span a tree containing one root node and one non-root node. All measurements were carried out free of external network interruptions, only the loopback device was used. Two measurements were taken; one showing the native GRIDKIT framework delivering

the messages without any filtering or tagging and one showing the performance of the GRIDKIT framework using the prototype presented in this section. **Table 6** below shows the delay measured per message of the second set of 1000 sent messages.

Measurement	Mean Msg. Delay in s	Std Deviation in s (%)	No. of Msgs. Sent	No. of Msgs. Received (%)
GRIDKIT & Prototype	0.058	0.013 (22)	1000	800 (80)
GRIDKIT	0.055	0.013 (23)	1000	1000 (100)

Table 6 Quantifying the Overhead of Virtual Overlays

It is evident that the virtual overlay firewall performed its intended purpose since the expected number of messages (200 or 20%) did not reach the receiving node in the test using the prototype implementation. It also shows that all messages send using the native GRIDKIT implementation arrived at the identical receiving node. The difference between the average message delay when using the virtual overlay compared to not using it is around 0.003s. The standard deviation calculated for both measures is 13ms. In all instances, the measured delay from all messages sent using the altered framework was smaller than delays measured for the pure GRIDKIT implementation. This might indicate that the overhead added by the prototype implementation is smaller than other factors interfering with the message transmission. The main suspected factors are the virtual machine internal thread management as well as the process scheduling of the operating system.
This experiment shows a sample application that realizes the basic architecture of a virtual overlay. The performance metrics detailed above prove that the configurability and expressiveness that is achievable using the proposed system makes the overhead insignificant. The next two experiments build on the implementation presented above to present more complex scenarios.

4.2 An Enhanced Virtual Overlay

Overview
This experiment aims at showing that our approach is not limited to message filtering or static routes but that virtual overlays can use their own state and a meta-routing algorithm to reproduce GRIDKIT's Control, Forward and State overlay pattern introduced in **section 3**.
As illustrated in **Figure 7** below, the namespace spanned by a Chord ring is used to globally address messages while messages sent to nodes in the Chord ring are actually routed via appropriate direct routes, which are in this example smaller Chord rings (with only 2 nodes).

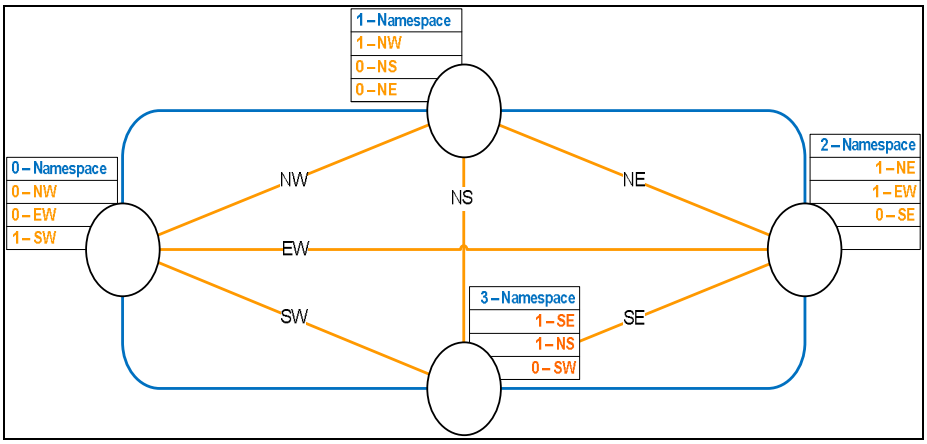

Figure 7 Illustration of TBCP Trees and Chord Ring Setup

This gives developers the flexibility to create virtual overlays which do not route messages within the actual overlays but between multiple overlays.

Implementation

As shown in **Figure 8** below, central to achieving this experiment's aim is the Routing Packet Overlay which uses a time controlled trigger to send information about each node it is deployed on in intervals. If the trigger fires, the Routing Packet Overlay obtains a list of all local Chord endpoints created by the Multi CHORD Overlay component. It then creates a message containing the global identifier (defined for the global namespace Chord ring) of the current node as well as the name and local identifier of the node in each Chord Overlay. The packet is sent via the local network named within the message. Thus the nodes in each local network can route messages correctly as they get to 'know' the global identifiers of the nodes on that network. This data is stored and managed in Routing Table.

The Multi CHORD Overlay component was developed to provide a unified interface to a multitude of Chord overlay networks. The component uses a network name to distinguish between the Chord overlay networks. In order to create a prototype which shows easily verifiable behavior, the Chord framework implementation was altered to support direct definition of node IDs.

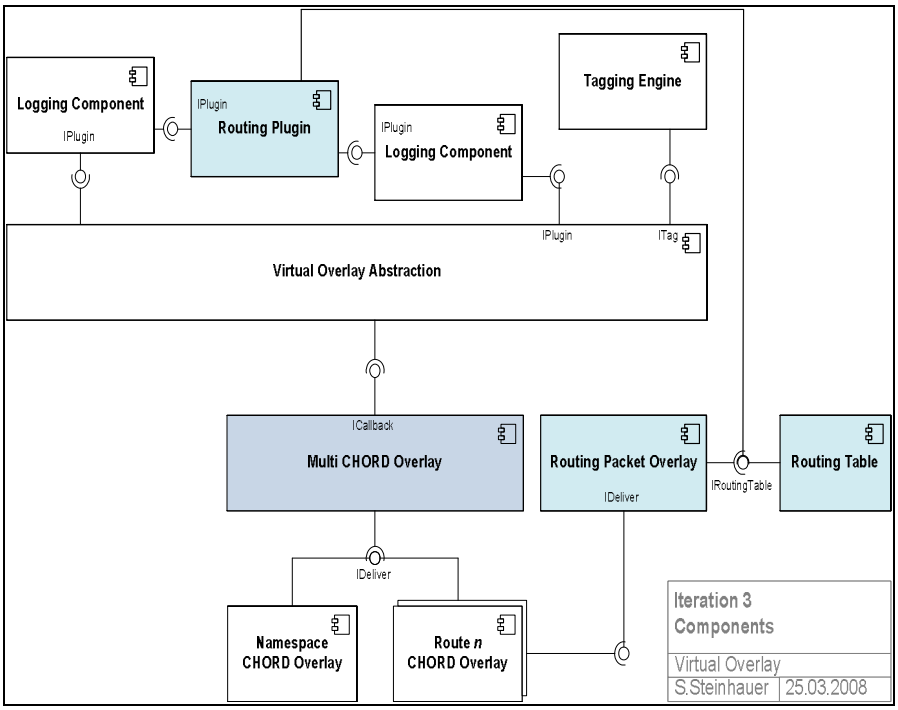

Figure 8 Major components in the scenario "Enhanced Virtual Overlay"

Figure 8 above shows three major components developed as part of this prototype (Routing Packet Overlay, Routing Table and Routing Plugin).

Evaluation

This scenario shows how the given components work together to route messages in a multi-overlay environment through collaborative message routing between nodes. The evaluation scenario is based on an overlay network environment with two nodes being part of two different Chord networks. Since the overhead of creating and maintaining the routing data within the hybrid network significantly influences the performance, it is not quantified. A scenario to demonstrate the effectiveness of the proposed system in an expressive manner would have required a bigger network and setup work beyond the scope of this paper.

5 Related Work

In Cooper et al's paper 'Trading Off Resources between overlapping Overlays' [3], an architecture called ODIN-S is introduced which has a focus on different methods to mediate resource usage between coexisting overlay networks. It uses a set of ingoing and outgoing filter to intercept messages on a shared communication layer. In this approach overlay networks are not stand-alone entities but plugins running on top of a common transport system. This transport system communicates with a set of filters in

order to control throughput and order of messages being sent through the overlay network. ODIN-S also assumes a homogeneous deployment of ODIN-S instances since it uses specific receiver originated messages to control the throughput of messages on sending nodes. In general the paper shows that manipulation of messages on their entry point into the overlay environment can be used to achieve QoS through the control of resource conflicts between coexisting overlay networks. Some generic ideas for the design of the proposed system were inspired by a project called P2 [2]. This project makes use of a declarative language to define overlays on top of a shared transport layer. The work stresses that declarative approaches can be efficient and expressive for describing the behavior of overlay networks. The novelty of the concept of virtual overlays, as detailed in this paper is that it addresses the more general management of collaboration and competition between multiple overlay structures.

6 Conclusion

This paper has presented an argument for the use of *virtual overlays* as a technique by which competition and collaboration between co-existing overlay network structures can be managed. Although a number of middleware frameworks e.g. GRIDKIT [1] and P2 [2] currently offer support for the co-existence of overlay networks, co-existing overlay networks inevitably interfere with each other's performance either through competition for resources or the lack of collaboration between them. More specifically, the paper has provided a high level overview of a middleware design which uses a meta-overlay to combine the strengths of multiple overlapping overlays (hybrid overlay networks) with a strong focus on dynamic adaptability, flexibility and configurability. We therefore argue that the use of virtual overlays to resolve resource conflicts, optimize performance via collaboration between multiple overlay structures and provide a higher-level abstraction that gives developers control over the overlay networks they deploy is the way forward in the design of next generation middleware. Areas of future work include research into deployment of multiple rule sets, development of a custom rule engine and rule language, self-configuration of rule sets and performance metrics in a large scale deployment environment.

References

1. Grace, P., G., C., Blair, G., Mathy, L., Yeung, W., K., Cai, W., et al: GRIDKIT: Pluggable Overlay Networks for Grid Computing. In Proceedings of Distributed Objects and Applications(DOA), Cyprus (2004)
2. Loo, B., T., Condie, T., Hellerstein, H., M., & Maniatis, P.: Implementing Declarative Overlays. In Proceedings of ACM Symposium on Operating System Principles 2005 (SOSP), Brighton, UK (2005)

3. Cooper, F., B.,: Trading Off Resources between overlapping Overlays. In Proceedings of the ACM/IFIP/USENIX 7th Middleware Conference, Melbourne, Australia (2006).
4. Aberer, K., Alima, L., O., Ghodsi, A., Girdzijauskas, S., Haridi, S., & Hauswirth, M.: The essence of P2P: A Reference Architecture for Overlay Networks. In Proceedings of the 5th IEEE Conference on Peer-to-Peer Computing, Konstanz, Germany (2005)
5. Stoica, I., Morris, R., Karger, D., Kaashoek, F., & Balakrsihnan, H.:Chord: A Scalable Peer-to-Peer Lookup Service for Internet Applications. In Proceedings of the ACM SIGCOMM Conference, San Diego, CA, USA (2001)
6. Davis, A., M.: Operational Prototyping: A New Development Approach. IEEE Software, 9(5), pp 70-78 (1992)
7. Rowstron, A., & Druschel, P.: Pastry: Scalable Decentralized Object Location and Routing for Large Scale Peer-to-Peer Systems. Lecture Notes in Computer Science, 2218, 329 et sqq
8. Mathy, L., Canonico, R., & Hutchinson, D.: An Overlay Tree building Control Protocol. In Proceedings of the 3rd International Workshop on Networked Group Communication, NGC, London (2001)
9. Coulson, G., Blair, G., Grace, P., & Joolia, A.: A Component Model for Building Systems Software. In Proceedings of IASTED Software Engineering and Applications (SEA), Cambridge, MA, USA (2004)
10. The Log4J Appender, http://logging.apache.org/log4j/docs/index.html
11. Conoboy, B., & Fichtner, E.,: IP Filter Based Firewalls HOWTO Tutorial, http://www.obfuscation.org/ipf, 2002
12. IP Filter ver. 4.1.27, http://coombs.anu.edu.au/~avalon/
13. Toussaint, A.: Editor, Java Rule Engine API: JSR-94, Java Community Press, Sept. 2003
14. Friedman-Hill, E.: Jess Information, the Jess Engine for the Java Platform, http://www.jessrules.com/jess/index.shtml
15. JBoss: JBoss Rules Documentation Library, jboss.org: http://labs.jboss.com/jbossrules/docs
16. Hammurapi Group: Hammurapi Rules, http://www.hammurapi.biz/hammurapi-biz/ef/xmenu/products/hammurapirules/index.html