JVAR: Java Virtual Active Routing

Diplomarbeit
der Philosophisch-naturwissenschaftlichen Fakultät
der Universität Bern

vorgelegt von
Marc Brogle
2004

Leiter der Arbeit:
Prof. Dr. Torsten Braun

Forschungsgruppe Rechnernetze und Verteilte Systeme (RVS)
Institut für Informatik und angewandte Mathematik

JVAR: Java Virtual Active Routing
Diploma Work 2004

Marc Brogle
RVS-Group
IAM, University of Berne, Switzerland

April 2, 2004

Abstract

This document describes an implementation of a virtual active routing framework written in Java. First an overview over classical networks compared to active networks is given. Possible applications of active networking are then presented. Following that the idea behind the framework and the technical aspects and implementation are discussed. Some examples realised with the implemented framework are explained in detail as well as some performance issues are evaluated. An outlook with possible extensions of the framework and a summary conclude the document. The appendices contain more detailed information about the programmed classes with additional source code examples and a closer look at the used logging framework.

The diploma work was under the lead of Prof. Dr. Torsten Braun
RVS-Group, IAM, University of Berne, Switzerland
Duration: April 21st 2003 till April 2nd 2004

Contents

List of Figures

List of Tables

1 Introduction to active networking

1.1 Traditional passive networks

Classical networks deliver data, or more specific, packets from a sender to a receiver. The stations (routers) in the network which these packets pass through are simple store and forward points. These routers just examine the packet headers, and then forward these packets according to a predefined routing table. This table tells the router the next point in the network for the packet to visit. The packet will be handed from one router to another, until the recipient of the packet has been reached. This means that the recipient is in a sub-network, which is directly connected to a router. In this case this would be the last router on the route from the sender to the recipient. So these passive routers hand the packet from one network to another via ports or links of the router without modifying the contents of the packet. See also figure 1 on page 2 which shows a traditional router and network setup through which a data packet is being sent by the store-and-forward method.

1.2 What is active networking?

Active networking [Tiw00] is a relatively new idea, in which routers are not only simple store-and-forward points for network packets. In an active network, routers or switches perform actions on the packets which go through them.

It is possible to put certain special programs into the network to configure and re-program the routers in the specific active network. Two main approaches to the realisation of active networking exist. First, there is the programmable switch (or out-of-band) approach, where the packet format is maintained, and a mechanism for downloading and triggering (active) programs is provided. The other is the capsule (or in-band) approach, where packets are replaced by capsules. These capsules contain small programs that are executed in the active nodes (routers) in the network. The possible actions these small programs contained in the capsule can cause, are only limited by the programming language used and the creativity of the implementer. Though a limitation of this approach is that the program size carried in the capsule should not be too big. Since the packet size in a network is limited they should for example be less then 1 Kilobyte). See also figure 2 on page 3.

Also resource management has to be taken into account. There are two categories of active network applications. The first one only deals with header processing, the second one does also some processing on the payload itself. Header process-

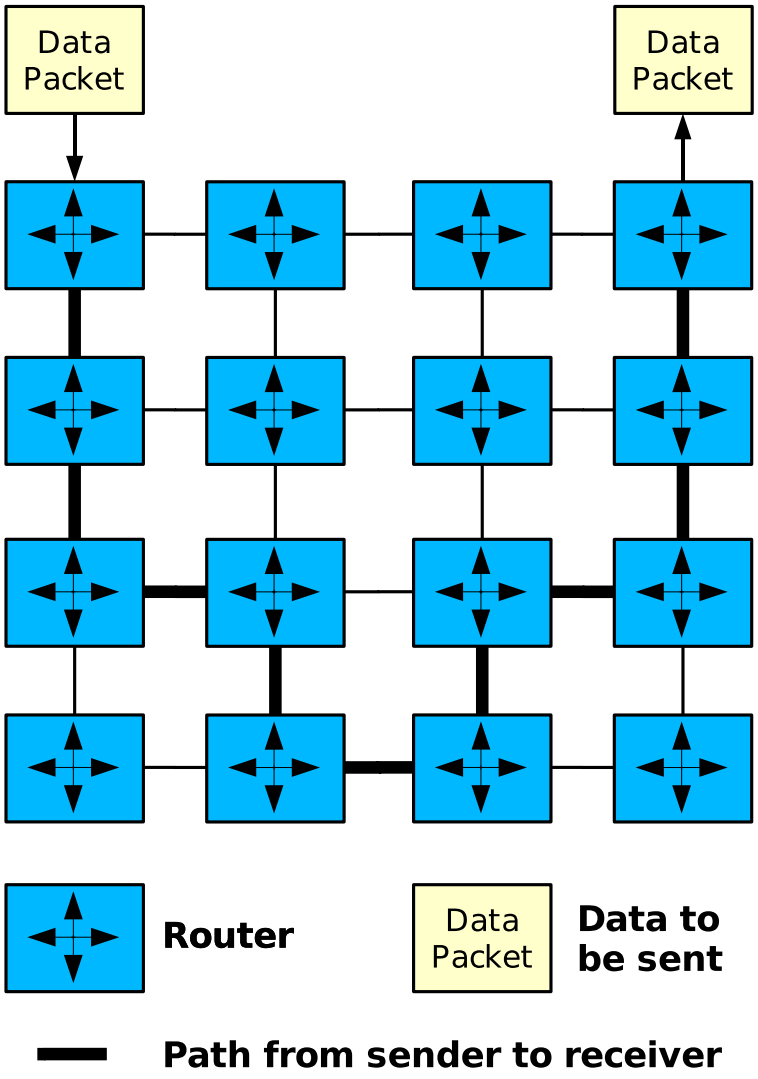

Figure 1: Traditional passive network example

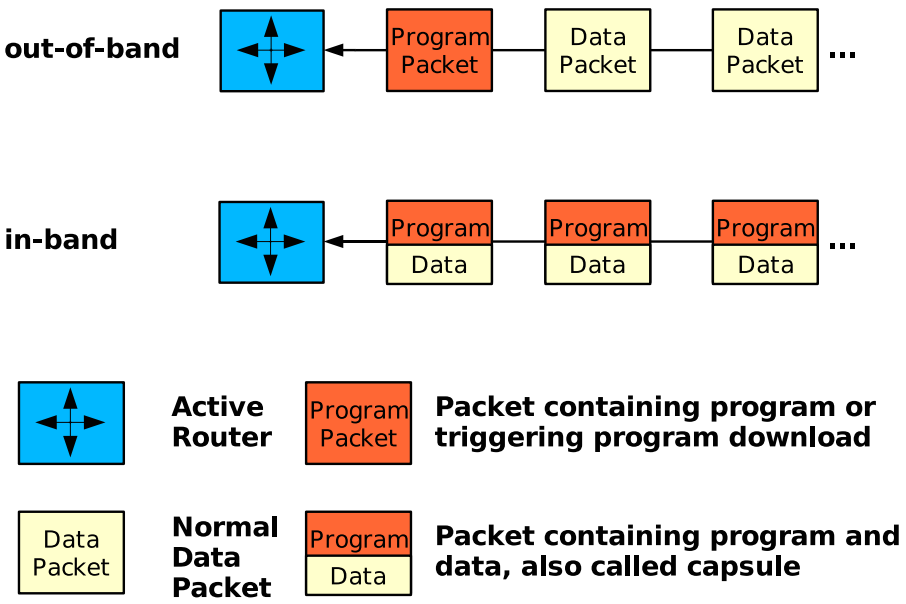

Figure 2: out-of-band and in-band approach

ing applications are restricted to read and write in the header of the packet, thus the processing complexity is independent of the packet size. On the other hand, the payload processing applications are allowed to read and write all the data in a packet. The complexity of an active application is increased if the payload processing approach is used. To use the payload applications efficiently and safely, the resources of the framework and also the router itself have to be managed. Active nodes resource management includes managing bandwidth usage, free or used memory of the router as well as the resources of the router's processing unit (CPU).

To have a better understanding of active networking and the actions mentioned earlier, some possible active networking applications will be presented. This diploma work wants to provide a simple framework, in which some of those applications could be realised very easily. All presented example applications of active networking can be realised in a real network or in an virtual network with virtual routers (see also chapter 1.8 on page 16). These examples could be realised either with the in-band or the out-of-band approach. Some of the example applications could even be realised with both approaches. In the chapter 2 the problems and limitations of both approaches will be discussed. A good general overview to active networking can be found in [TSS+97] or in [Hua02].

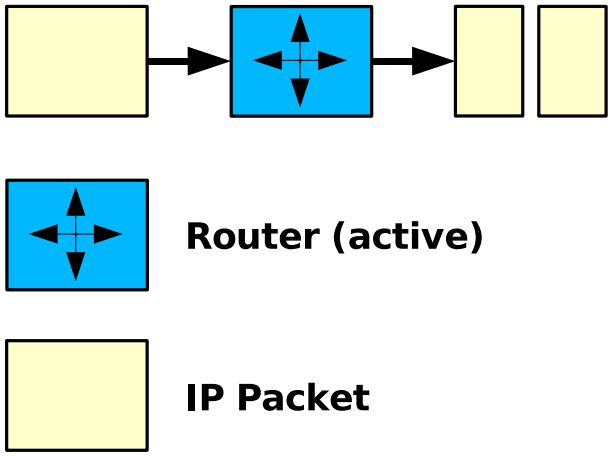

Figure 3: Packet splitting / duplication

1.3 Routing based active networking example applications

Basic actions for these examples include to duplicate packets or even make a certain number of copies of them. This can be used by multicast[1] applications. The decision of how and where to duplicate is done by the packets themselves. They setup a prior rule matching their stream, which will duplicate the packets by some additional information the packets provide. This information can for example be directly contained in their payload. These actions can be realised by using either the in-band or out-of-band approach as well as using them both in a mixed form.

Other possible actions include splitting a packet into smaller sub-packets (see also figure 3 on page 4). These split packets can be sent through alternate routes to increase overall performance or to make the stream more reliable. These actions can be realised by using either the in-band or out-of-band approach as well as using them both in a mixed form.

Explicit Multicast

Packets can be sent to reach multiple users which is called multicasting. Explicit multicast ([BFI$^+$] and [BL]) differs by not sending a packet to a group address but by having all the recipients addresses stored within the packet. The packets or its programs decide on each router, if there is a client that might be interested in the data that the packet holds in the attached subnet. Also some receivers of the packet could have to be reached by separate links from this router on. Then copies of the

[1] Multicast: same kind of information for more than one recipient in only one packet

packet are being created by the packet itself or by a program preloaded prior to the sending of the explicit multicast stream. These copies are then sent on different links further through the network. This might happen multiple times until all the possible receivers have been reached.

Explicit Routing

Explicit routing of packets helps to provide guaranteed bandwidth or response times. Instead of using the predefined route defined by the routing tables in the router the packets themselves or the programs matching the stream decide what route they want to take. This can be decided by asking the router about bandwidth usage of its links as well as contacting agents. These agents which are also active network components in other routers can be asked about the bandwidth and other resources usage at their location. The requests for that information can be executed periodically which allows the programs or packets requesting the information to be served immediately.

Usage based rerouting over alternate routes

If some links of the router are heavily used, some packets could be routed over alternative routes to the same receiver. This helps to distribute bandwidth usage over the subnets laying in between more evenly. The packets get delivered and bandwidth usage of some links are kept low in order to have them ready for other clients with guaranteed bandwidth.

If some links have faster response time and the sender has an agreement with the network service provider like mentioned above (for example for a maximum packet delivery time), this service can be guaranteed by changing the standard routes on-the-fly for certain flows. This would assure a short delivery time if some routes are heavily used or their response time would be too low for the requested service.

1.4 Examples of security related active networking applications

Tunnelling data streams

To reduce the bandwidth usage in a network, specific data streams being sent through a network, can be put through a GZIP[2] tunnel. This means, that at a certain point in the network, the payload of a packet gets compressed with the GZIP algorithm. From this point on, the packet normally is reduced in size and

[2]GZIP: freely available compression algorithm

can be passed faster through the network. It is then passed on, until it reaches a point in the network, where the payload will be decompressed. From there the packet passes on in its original form as it entered the network before being compressed. This way the whole compression / decompression and tunnelling is completely transparent. Neither the sender nor the receiver know that the packet has been manipulated. See also figure 4 on page 7.

Changing encryption strength

The encryption strength could be changed on-the-fly in the active routers. The strength for example could be changed from an encryption with a 64 bit key to an encryption with a 256 bit key. The decision would be made depending on the free CPU resources, the encryption algorithm or the key strength, which a router can provide (see also figure 5 on page 8). Therefore, on heavy traffic and not much CPU resources available, the overall encryption could be lowered for some packets. The decision could be made directly by the packets themselves which provide the algorithm for the decision making process (in-band). On the other hand the decision could be made by a predefined rule that would have been setup prior to the stream transmission. Then a rule matching certain streams or resource capacity conditions would make the decision (out-of-band).

Secure pathways

If the trust in a network (and the parties in between) is doubtful, a secure pathway through the network can be created on demand. A packet could enter a network asking for at least being encrypted by a 128 bit key. Therefore it has to be routed through the network using only the safe routers on a pathway providing the desired strength of encryption and security. The packet could become encrypted by for example 3DES [Cas] or Blowfish [Ane01] algorithms. Once the packet has been encrypted, only routers supporting this encryption method and strength will be able to handle the packet correctly. The whole encryption / decryption happens completely hidden from the user, which will receive the packet without knowing, that on a certain path on the way it has been encrypted and decrypted. See also figure 6 on page 9.

Sending encrypted and split packets over two different routes

Packets could be split into two separately encrypted packets (only the payload would be encrypted), which then are sent over two different routes ([Bro00], [GBBb] and [GBBa]). The packets would travel on separate paths and towards the end reach together a certain point, where the original data (one packet) would

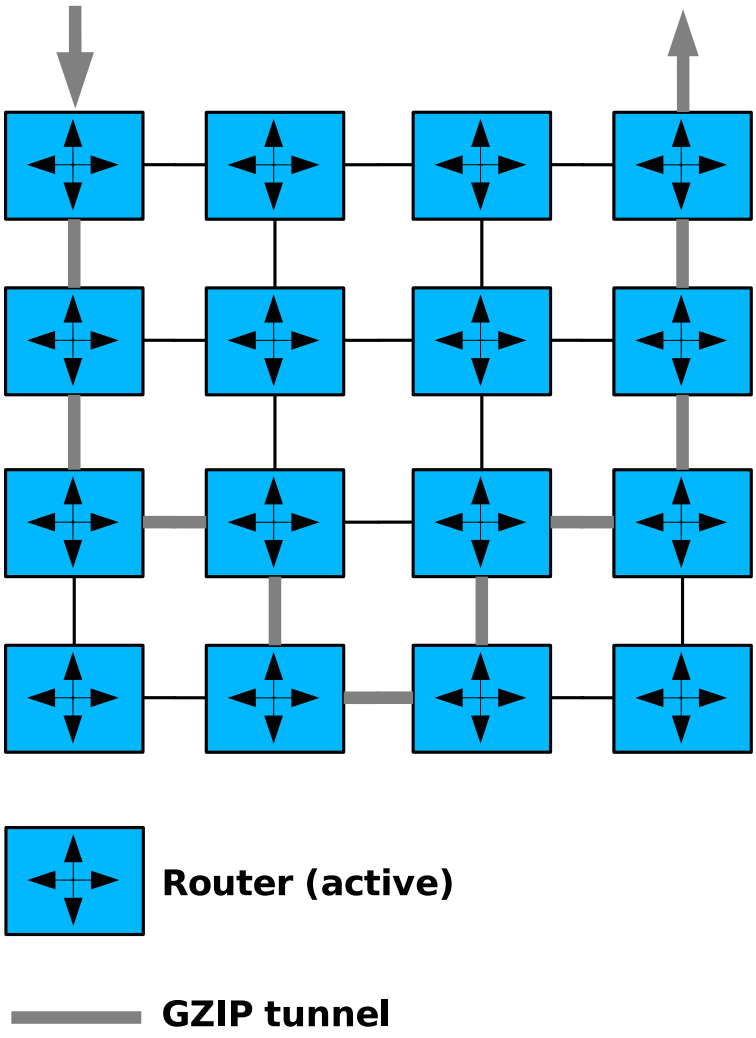

Figure 4: GZIP tunnel through a network

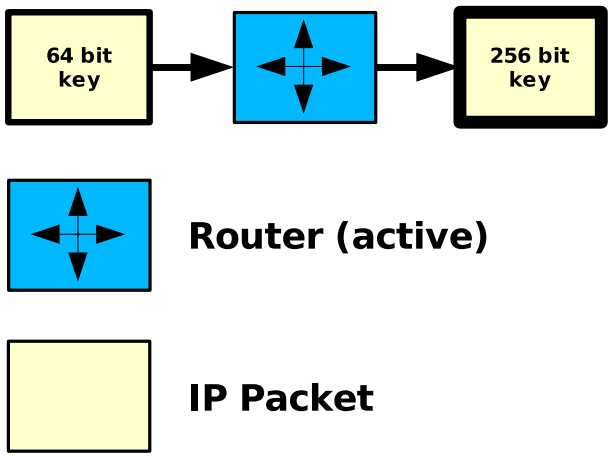

Figure 5: Encryption strength conversion

be restored again and continue its way to the target. Both of the packets are needed to reconstruct the original data contained. If someone intercepts the stream and tries to figure out the data contained, the original data cannot be restored. The interceptor would miss the other part needed which is sent over a complete different route through the world wide network. See also figure 7 on page 10.

1.5 Multimedia active networking example applications

Payload in general can be modified by the packets themselves on their way through a network. There are various kinds of modifications that can be applied by the algorithms provided by the packet (in-band) or by preset rules and modification filters (out-of-band). One of the possible modifications include conversion of the payload to another format. While an image is sent through a network, its format for example can be changed from GIF to JPEG (see also figure 8 on page 11). Information on video streaming in networks can be found in [BGB] and [Bal].

Basic image modification on flow usage and free capacities in routers

To make video conferencing or live camera feeds more reliable, the images sent over the network could be modified on certain conditions. Since no dropped frames should occur the quality of the image gets reduced, which will lead to smaller packets and less information to transmit. This works fine if the quality of the image is secondary but it is more important to send images without loosing them at all. The images of certain formats (JPEG, GIF) are being sent through the network. Periodically or on each packet containing such an image the status

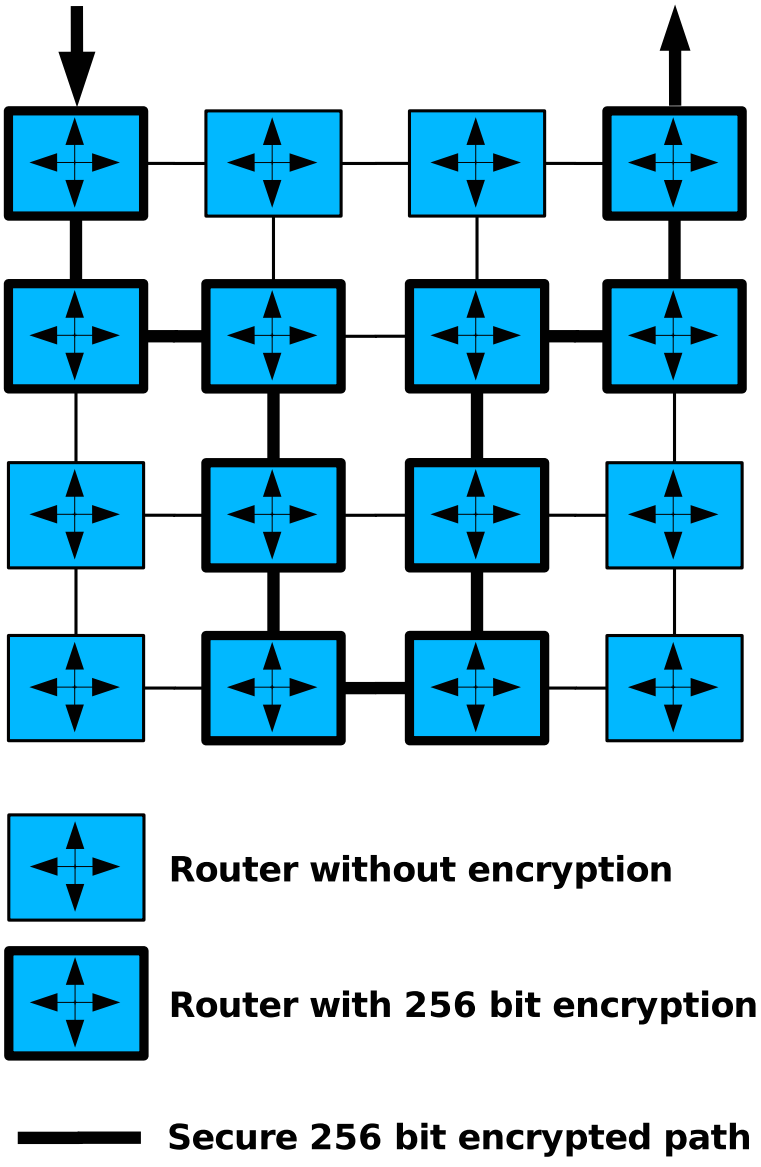

Figure 6: Encryption tunnel through a network

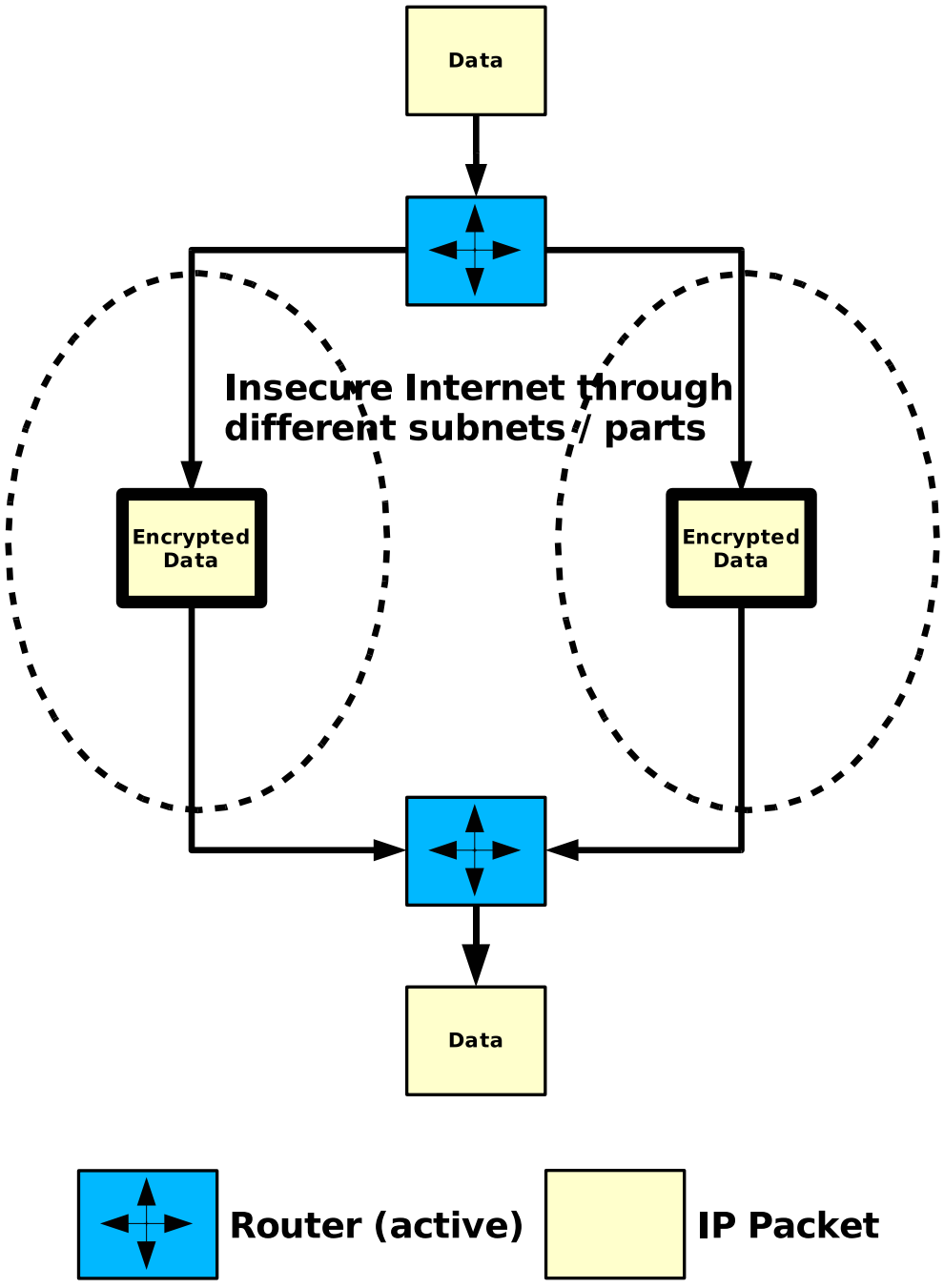

Figure 7: Example of splitting and encrypting packets

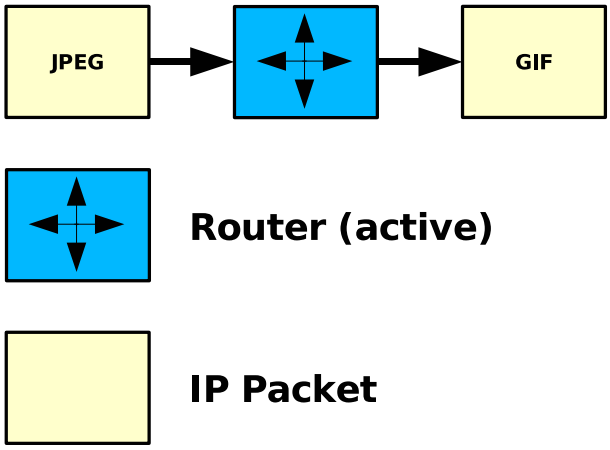

Figure 8: Type conversion

of the resources of the router are being requested. This information could include CPU usage, router link usage, overall bandwidth usage, free memory resources, general QOS information, link speeds or information of other routers attached to the links of this router. Therefore the modifications of the image will be evaluated regarding the mentioned information and then the decision is done, how much or if at all images have to be modified. Modifications could include changing the compression factor (for example JPEG quality), reducing the colour depth (for example 24 to 8 bit), conversion to black and white, resizing the image (80%, 50%, etc.) or even some combinations of those.

Video-split

Frames of a video stream (for example from a video conference or a movie being played) can be "split" into different quality streams. This splitting would happen on critical nodes in the network pathways. Since the active components on the routers have information about the speed or restrictions of their links, some pathways might not be able to handle the full quality stream. The frames sent are then only distributed from such a critical node with a reduced quality (see also figure 9 on page 12).

Voice modifications

Voice streams (PCM, WAV, MP3) could be reduced in quality (for example 128 kbit/s to 64 kbit/s MP3), depending on the network bandwidth resources, that are available at the moment. On each packet this decision can be done, so it is an in-

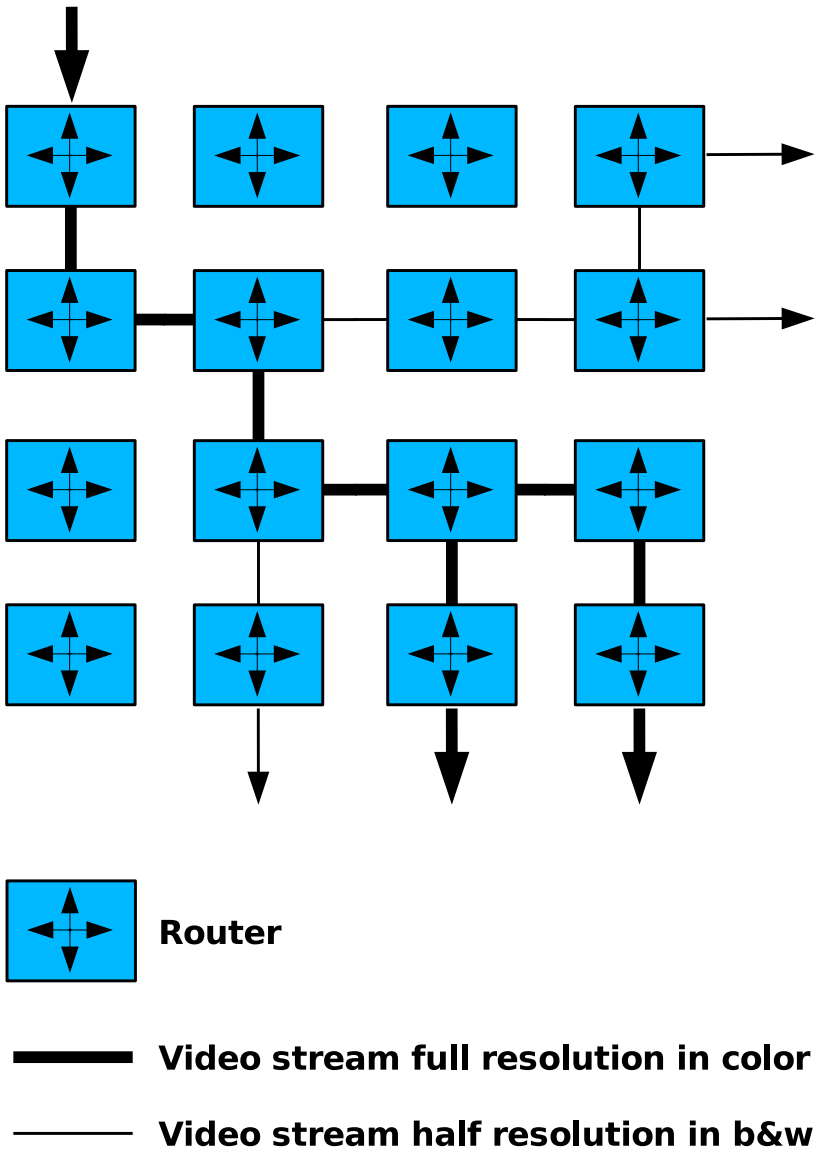

Figure 9: Video split in a network

stant adaptation to the current situation. This would work for VoIP (Voice over IP) as well as for any other live sound streams (Internet radio, chat programs, etc.). If the network is at high usage or at its limits, still all information will be sent. No packet drops would occur and only a bad voice quality will result out of the taken modifications. It is more important to get the whole spoken sentence (even if in a bad quality), than some words would be missing of the spoken sentence.

The voice streams could also be sent with an explicit multicast and voice protocol conversion points could be setup (PCM to MP3, WAV to WMA, etc.) for certain streams or on certain points. This could help to distribute one stream in different formats for different recipients in a network by combining the explicit multicast and the voice conversion points. Instead of sending many different streams directly, the other streams in a different voice format could be created on-the-fly at the needed points in the network. From this point on after or before the packets are split up they would be sent further through the network, till all recipients would be provided as needed.

1.6 Network quality active networking application examples

Packet dropping

An active router can drop packets [YNR98]. The decision is not made by the router itself, but more by the packets themselves (in-band) or a special task matching a certain stream that has been setup before by another packet (out-of-band). After that, for example all late or as less important tagged packets get dropped on certain conditions. These could include high bandwidth usage in the network around the router, to which the packets will be forwarded, so it cannot support any additional packets to be processed.

The selection of what to drop is really made by an algorithm or a condition the stream itself provides to the router (either in-band or out-of-band). After a certain time, this rule or algorithm can be erased or modified by the stream itself (out-of-band). The capsules could contain no more or different drop-algorithms and rules (in-band).

Packet buffering

Another possibility could be the buffering of packets in a router [GKPR], which can be decided as well by the packets directly. They for example would have to wait for some other packets to arrive first at this point before the travel to the next router would go on. The buffer time and rules how or for what to wait can be

decided directly by each packet (in-band) or these rules could be setup ahead by some packets containing the rules for the following packets on that stream (out-of-band).

Flow based payload compression

Instead of setting up fixed GZIP algorithms as described before, the compression factor can be adapted depending the bandwidth usage in the whole network or the free CPU resources of the router. Alternative compression algorithms could be used that provide different types of compression factors like RAR [Rar] or other ZIP algorithms. These actions will help to reduce bandwidth usage, if the free bandwidth gets to a critical level. Instead of dropping the packets (if for example the network provider does not guarantee packet delivery) they get compressed and sent through the network as long as there are some little free resources left.

Agent notification of flow / usage / status

Agents ([Gün01] and [dMCPT]) could be setup on the active router components which could request and send status information about their location. This information could include CPU usage, free resources (memory, etc.), bandwidth usage, link status of the router to which they are connected to, running programs and actual rules on certain flows. Depending on this information provided, programs or rules that are running on the other nodes can decide how to reroute or to modify a certain packet. Instead of asking the other involved active router components for these informations directly, the rules or programs can ask the agents running on the same node they reside on. These agents already might know the information. The agents would provide a well defined interface to get the information. This way all the different applications described earlier could ask the requested information from the agents by the well defined interface instead of having each of them their own way of getting this remote information.

Redundancy reduction of stream casts

If for example two clients receive the same stream, which is sent as two separate streams, the router can make a packet content comparison. If they are in a certain time frame, these two packets could be put together to one explicit multicast packet as described earlier. This packet would then then be sent through the network and split up again to the two original packets as needed. This could be used in slow networks or when a high bandwidth usage occurs. Important issues to consider would be to use just a little queue, which holds back packets for a certain short time only. If after a certain time no duplicates would be detected in

the queue, the packet should be sent as it is with high priority. Since through this hold back in a queue a certain delay occurs, this delay has to be quite short if the redundancy reduction is applied to real time streams (audio and / or video). This detection should not be done on every node, since the delay gets augmented on every redundancy checking node a packet passes through and gets queued up.

Classical TOS handling, QOS and other services

Also classical TOS (Type of Service) and QOS (Quality of Service) handling can be provided by the active components ([Bau02] and [BBb]). Like mentioned above bandwidth can be guaranteed (for example always 128 kbit/s dedicated for certain flows). Also minimum response times can be assured (not more than 100ms delay till it arrives at the receiver). Burst rates could be optimised (trying to get as much as possible burst rate available). Packet priorities could be handled or changed. Packets with lower priority would get delayed, to guarantee fast handling of high priority packets. Finally real time streams would also be supported and explicit multicast could be provided.

1.7 Content analysing active networking example applications

Unwanted words in the packet's payload can be removed for example with the help of a word blacklist. Also a virus check of the payload can be done at some designated virus checking points. Content analysing active networking applications can be realised with the out-of-bound approach.

Parental rated payload modification

Text payload streams could get analysed, and if any offensive words are contained, they get either replaced, censored or deleted. Also images could be analysed by certain algorithms for unsuitable content. This can either be done by analysing keywords in the image header or by image content recognition algorithms. It could even be applied on voice streams if some voice recognition algorithms exist that would recognise these offencive words, which then could be censored or modified. Also HTTP (Hypertext Transfer Protocol) requests or responses could be dropped, if they contain offencive or not for minors suitable content.

On-the-fly text translation or correction

Text being sent through a network could be checked for writing errors (for certain streams) on-the-fly at certain designated checking points in a network. This could be done for example by using the Ispell [Kue] program or other libraries provided.

Also on-the-fly translation of text for example in UDP payload of chat messages could be done on these designated points. This could be realised by setting up a rule on a certain stream from a sender to a receiver that might speak different languages. Only real text payload and not additional headers of for example the chat programs should be translated or corrected. Translating or correcting the chat protocol header would make the packet unrecognisable for the chat program.

1.8 Virtual routers for network emulation

Virtual routers [BBKW] simulate a real router implementation. They are little programs or services running on a machine. They provide the basic router functionality without actually being a real router. They do not use the real network of the computer that executes them.

Virtual routers [BBc] can be connected together in order to simulate a whole router setup of a big network. These virtual routers behave like they would exist in a real network, providing the routing functionality by the routing tables implemented in these virtual routers. Only the architecture and the resources of the used machine limit the amount of routers that can be simulated on one computer at a time.

These virtual routers and virtual networks [BBa] help to use and test the JVAR framework without actually having a real big network infrastructure. Changes and new functionalities can be tested directly without having the fear of breaking a running system or disabling a stable running network by introducing the new components.

It is also possible to use different computers and span the virtual routers over those connecting them together virtually through the existing real network between the used computers. Instead of only having one virtual router per computer, it is also possible to setup complete virtual networks on one computer. These virtual networks can then be connect through the real network between the computers to other virtual networks that run on other computers. See also figure 10 on page 18 which shows two computers running each a virtual network with virtual routers and where the two virtual networks are connected through the real network.

The different virtual routers used are completely transparent to the underlying system. It is even possible to have an entry point from a real network to a virtual router setup with a lot of virtual routers included. A packet from the real network enters the virtual network at the mentioned entry point and after it has been routed through this virtual network it can exit the virtual network and re-enter the real network at a designated exit point in the virtual network.

Generally it is possible to simulate complex behaviour and different applications with the help of the virtual routers and the virtual networks. On top of the virtual routers, active networking implementations can be added. It is then possible to simulate a whole active networking setup without having of a lot of machines or real routers.

The active networking implementations provide basic interfaces to communicate with the virtual routers. The packets can be taken out of the flow that passes through the virtual network. The extracted packets will be processed by the algorithms the packets provide themselves or by other preset rules. After the modifications they get injected back into the virtual network through the virtual router.

1.9 Existing Java active networking platforms

Different active networking platforms written in Java exist. ANEP [AB+], ANTS [Weter], JANOS [THL01] and ASPEE [BLBF] will be shortly presented. For a technical comparison of JVAR and other active networking frameworks see also chapter 2.5 on page 28.

ANEP

AENP is an interoperability layer for Active Networks. The platform is described on the ANEP webpage [WPA]:

> ANEP specifies a mechanism for encapsulating Active Network frames for transmission over different media. The suggested format allows use of an existing network infrastructure (such as IP or IPv6) or transmission over the link layer. In order to support ongoing research, the proposed mechanism is as generic and extensible as possible. This mechanism allows co-existence of different execution environments and proper demultiplexing of received packets.

ANEP is the work of many institutions currently researching active networks. The project started like ANTS in the early years of active networking with Java.

ANTS

The ANTS webpage [Weter] describes the toolkit:

> ants is a Java-based toolkit for experimenting with active networks. It provides a node runtime that can participate in an active network,

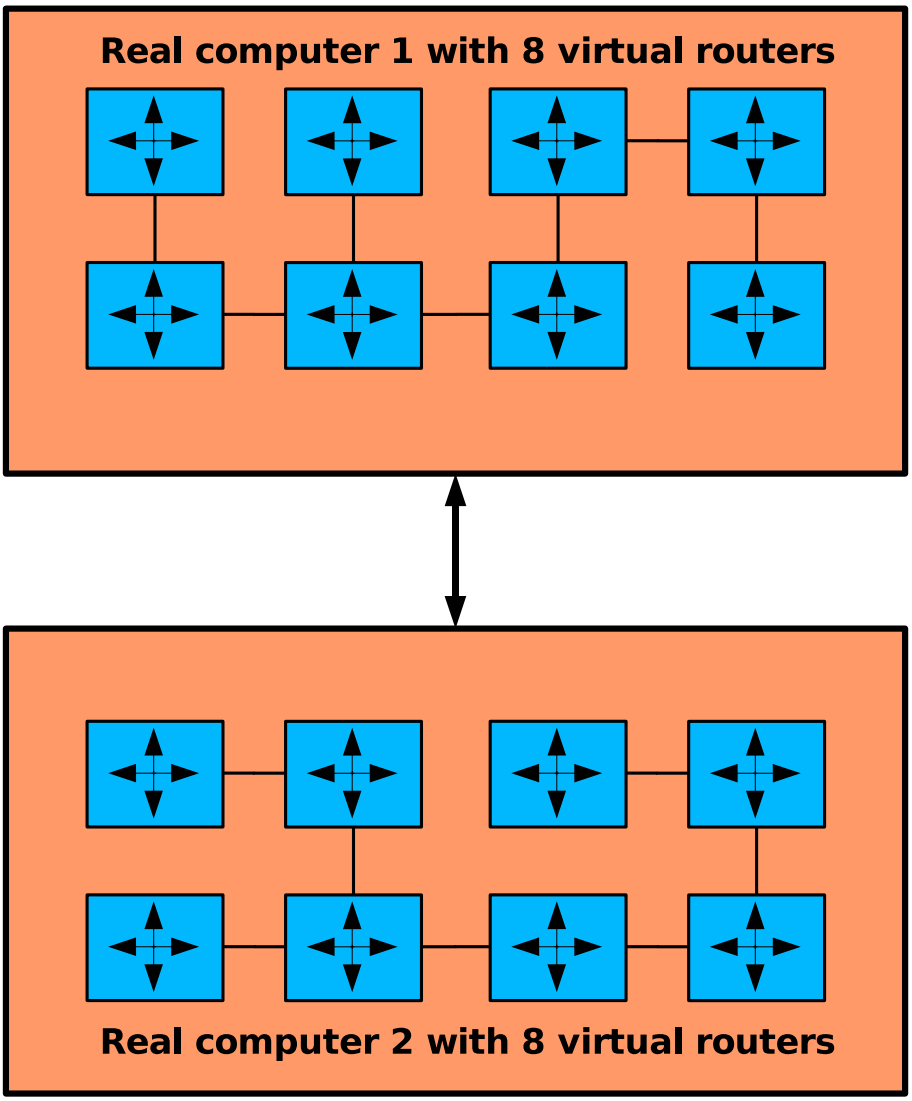

Figure 10: Example of a virtual router setup on different computers

> and a protocol programming model that allows users to customize the forwarding of their packets.

ANTS was one of the first available active networking platforms written in Java. Different new versions of ANTS exist which are often using JANOS to access the network.

JANOS

The authors of JANOS describe their project in [THL01]:

> Janos is an operating system for active network nodes whose primary focus is strong resource management and control of untrusted active applications written in Java. Janos includes the three major components of a Java-based active network operating system: the low-level NodeOS, a resource-aware Java Virtual Machine, and an active network protocol execution environment. Each of these components is separately usable.

JANOS is a very well maintained active networking environment. A lot of other active networking projects rely on JANOS or some of its sub-projects.

ASPEE

In [BLBF] the authors describe ASPEE:

> [...] the ASP Execution Environment (EE), a prototype general-purpose active network execution environment that initiates and controls the execution of Java-based active applications. Features of the ASP EE include support for persistent active applications, finegrained network I/O control, security, resource protection, and timing services.

1.10 Structure of the document

This chapter gave a short overview on active networking. Differences between traditional networks and the active networking world have been explained. The presentation of virtual routers and possible applications concluded the chapter together with this overview over the document.

In the following chapter 2 the JVAR framework, its usage and some other aspects of it will be presented in more detail. The chapter will explain the approach chosen for the JVAR framework. A discussion on the advantages of Java and the

JUnit framework will be presented. Comparison to other active networking solutions will be followed by the technical realisation of the JVAR framework. The different kind of components included, their duties as well as some short code example of how easy the framework can be used will demonstrated. Also some security issues, how they have been solved and the advantages of using a logging framework will be presented.

The usage of the JVAR framework will be exemplified in chapter 3. Different applications will be explained, and the solution using JVAR will be provided. All these examples are also fully programmed and can be used when starting the corresponding scripts that come with the JVAR framework. Performance measurements will conclude the chapter.

An outlook of possible extensions will be discussed in chapter 4. Extensions of basic packet manipulation as well as security issues will be examined. Possible performance accelerations will also be discussed.

For more technical issues, the appendices provide a complete class overview, as well as some more code examples that help to understand the JVAR framework. Also the usage of the logging framework will be presented in the appendixes.

2 The JVAR approach

2.1 Overview

JVAR is an active networking framework, that can be used with different kind of routers. The routers only have to support some basic communication possibilities. The framework makes it very easy to explore the active networking world.

The JVAR framework is simple to use and it is also very easy to understand the concepts used in it. It can be extended without much coding needed, because abstract[3] super classes provide all basic functionality and the application flow. Only the program parts or algorithms working on the data packets have to be implemented. Since there is also a separate layer for the communication with the router attached to the framework, almost any router providing a basic communication interface can be used.

The framework has been built as well to be used for simulations. An extensive configurable logging framework and GUI (Graphical User Interface) or console output information have been integrated. Therefore it is possible to follow every step while the framework is running in action and processing network packets.

The source code of the framework has been well documented. Javadoc[4] comments have been extensively used to help understanding the framework by browsing through the source code or the provided HTML overview. Also a lot of demos showing the use of the framework have been written, which helps to understand and simulate different kind of active networking applications.

The packet handling and packet header related operations have been written very efficiently. Therefore the basic operations, which are executed often do not reduce the overall performance of the framework. A pluggable Filter / FilterSet architecture has been chosen, which allows to easy integrate, change and extend possible actions on different kind of packet streams. The filters can be chained together with other filters to the so called filter queues.

A big focus was the platform in-dependency of the framework. This has been realised by using JAVA as the programming language for the framework.

---

[3] abstract classes cannot be instantiated, they can only be inherited from

[4] Javadoc: documentation standard for Java source code, that allows to generate automatic cross-referenced HTML documentation)

2.2 Why use JAVA?

Java [mic] does not seem to be the perfect candidate on first sight for system near manipulation of IP packets. But on closer observation a lot of advantages, which speak for the use of Java are found. Java is completely platform independent. So called virtual machines to run Java programs exist for almost every hardware platform, operating system and even for embedded devices. Java is easy to learn and widely spread. People are familiar with the language, which makes it reach wider target group to understand, extend and work with the framework.

It is possible to integrate native C and C++ code with JNI [Lia99] (Java Native Interface), which allows to integrate existing codes and algorithms into the framework very easily. JIT (Just in Time) compilers help to get higher performance over normal Java execution, since when executing the program the first time, it gets compiled to the platform it runs on. Thereby all platform specific features to optimise the overall performance of the system are used.

A lot of high level APIs (Application Programming Interface) for Java are available, which provide a large basis of additional frameworks and algorithms, that can be easily integrated into the existing framework. Good networking basic classes are already implemented like socket handling and higher level protocols (for example HTTP). Dynamic class loading supports loading unknown classes at start up time of the framework. During the execution life cycle of an application, other classes from remote locations can be loaded (even over the network) and then instantiated at runtime in the running program.

Classes can also be built and compiled during the runtime of the application and then integrated and used directly without having to restart the application.

Introspection of classes allows to detect new unknown methods at startup time. Also all features and methods of a class can be analysed this way. Thus it is possible to call methods, which are not provided by the super-classes and not known to the current instance scope.

The Java development and runtime environment is freely available and a very modern, object oriented programming language. A lot of freely available third party products for development, runtime optimisation and analysis exist. Java applets are runnable inside a web browser (Netscape, Mozilla, Opera, Konquerer, Internet Explorer, etc.) which could be used to access configuration programs for routers, or to get other statistical data of running applications.

There are also some disadvantages. Java does not always have a good performance. This mainly depends on the programming skills and as well on the availability of optimised JIT compilers for the platform it runs on. The IP manipulation is not directly available in the basic Java classes. The use of socket stream classes is not very flexible, since there are certain limitations inflicted by its usage. Adapting the configuration for different platforms can be time consuming. But still, the advantages that speak for the use of Java to create the JVAR framework dominate over the few disadvantages that could inflict on the usage of the framework.

To improve the performance or to avoid some other discussed disadvantages different kind of alternatives exist. Using JNI [Lia99] to access external platform specific optimised programs and algorithms would improve the performance of time critical applications. Also other programming languages optimised for certain specific applications could be integrated via bridges available for the Java platform. There exist are a lot of bridges or Java implementations of other programming languages like for example Python (Jython [PR02]), Tcl (Jacl), Ruby, Delphi and Haskell. Also the usage of special Java virtual machines and paradigms optimised for real time execution [BGB+00] and operating systems exist.

There exist several active networking projects that are realised in Java ([Weter], [AB+], [THL01] and [BLBF]). Java has become one of the favourite programming languages for active networking. [Tiw00] gives a good overview over different active networking implementations.

2.3 The beauty of JUnit

JUnit [GB] is a regression testing framework written by Erich Gamma and Kent Beck [Bec02]. It allows the developer to test the written source code by using unit tests for each class and its methods. These tests or even some tests put together to test suites are then run all together automatically. As a result, they will show if the classes and methods tested work as expected. Without automated testing, it is time consuming and difficult to ensure that changes will not break existing code.

The advantage for the JVAR framework by using unit tests is, that new components can be introduced and existing classes can be extended without the need to test the extensions or additions in a real environment. Existing tests can be re-run to ensure, that the added extensions did not break any existing code and functionality. The tests can be extended to assure also the reliability of the new functionality. Also new tests can be added to the framework to test the newly added components. All this helps to check if the newly added components work fine after integration into the JVAR framework.

Other active network implementations lack often this kind of "test before use" possibility. The new functionality has to be tested on real systems and also the assuredness that these extensions work as expected cannot be checked easily. The added components would have to be tested manually. This process has to be redone every time a faulty component has been detected. A long turn around when extending the active networking framework will then lead to less motivation of playing with the framework and extend and explore its possibilities.

With the simulated routers and the JUnit testing framework integrated in the JVAR framework, a stable active network implementation can be kept even when extending the existing functionality. Tests have to be written to check the new components incorporating with other existing components. These components also have to be tested with different input and the behaviour with non valid input has to be checked. The source code written for this diploma work provides a lot of simple example components. They all have according test example classes, which makes it easy to understand how to extend and add new tests to the framework.

2.4 The basic JVAR architecture

Basic node architecture overview

Generally a packet is sent from a client to another through a network by passing through some routers on its way. Packets traversing a router can get modified inside the active component attached to the router. The combination of router and its active component will be regarded as an active node. The mentioned modifications can vary and also other side actions might be triggered by this packet.

After a packet has been processed it might gets injected back into the network. The mentioned actions being triggered might execute tasks inside this node or notify other nodes to do different kind of tasks there as well. These tasks can include loading missing objects for this node or for the other nodes, notifying agents (locally or remotely) or even setup new rules and actions on certain streams trough this node or other nodes.

Now follows an overview with a short description of the main components that are used in a concrete active node using the JVAR framework.

A **Client** represents a real or simulated network client. Clients are connected to subnets, which are as well connected to routers. To simplify the overview and the examples, the clients have been connected directly to the routers.

Routers are real (or simulated) routers in a setup. Routers are interconnected with each other. They can have (to simplify the setup) clients directly connected to them. Routers have JVARCores connected (via RouterHandlers) to themselves.

The **RouterHandler** is the communication interface to and from the router for a JVARCore. It passes all data and requests to a JVARCore by a well defined interface and knows how to communicate with the special router implementation to that it is connected to.

The main component of a node is the **JVARCore**. It is the main active networking handler / execution unit, that matches actions to packets and also communicates with a router (through a RouterHandler) if needed. JVARCores can communicate with other JVARCores through the network by using the provided IP packets, which hold special tasks that have to be executed in the remote JVARCores.

To manipulate packets in a JVARCore, **FilterSets** (often several per JVARCore) are used. They are the active networking components, working on complete IP packets. The JVARCore delegates the IP packets to the corresponding FilterSets. FilterSets also use Filters to work on the IP packets. FilterSets can communicate directly with the Router (via the JVARCore) as well as creating new tasks to be executed locally or even remotely.

Filters (often several per JVARCore) are the simple components working on the IP payload only. Filters are used by FilterSets and never directly by a JVARCore. Filters cannot communicate with the Router and are only doing simple actions. Filters can be chained together (by a FilterSet) to build complex actions.

To observe the actions in a node, **Observers** can be attached to clients and JVARCores. These Observers are only used for the example active networking applications presented in chapter 3 to follow and see the active networking modifications. They display images or text, which are in the payload of a packet. They can be used with the GUI or just provide output information to the logging facility and console. Observers help to understand how the JVAR framework reacts in a network setup by watching the payload and actions happening in a JVARCore or client to which such an Observer is attached to.
Tasks are executable objects, which are passed between different remote JVARCores and executed in a JVARCore. Tasks are put into the TaskCore, which holds all tasks within the JVARCore. Tasks can be generated by JVARCores and FilterSets. They can be sent to remote JVARCores. They can be configured to be executed either once, several or even infinite times.

The **SocketClassLoader** in a JVARCore loads unknown active components from an (external) socket class server. These actions are triggered by tasks, which can either be created locally or can come from a remote location. SocketClassLoaders know from the task, from which specific SocketClassServer they should load the requested classes. **SocketClassServers** are specific implementations of class servers providing a source to load unknown but requested active components. Classes can be loaded by SocketClassLoaders. The SocketClassServers exist outside the JVAR framework and can exist on multiple locations. They have to be started aside a JVAR node setup.

The so called **MatchMaker** is a specific method inside the JVARCore, matching IP packets to TaskCores, Observers or FilterSets. After analysing the header data of an IP packet, it can find the corresponding class to hand the packet to. It will be either manipulated by a FilterSet, its contents be displayed by an observer or it will trigger some internal actions by a task in a TaskCore. Unlike Filters and FilterSets the MatchMaker does not manipulate the packet and its contents at all. The MatchMaker only finds the corresponding action represented by a Filter, FilterSet, Observer or TaskHandler that has to be executed on the packet. For an example of a basic node setup, see also figure 11 on page 27.

Packet flow

An IP packet passes through different stations on its way through a network and the active nodes. In this section, the packet flow inside a complete JVAR node setup (see also figure 12 on page 28) will be presented.

First the router receives an IP packet from the connected network. The JVARCore connected to the router (via the RouterHandler) asks the RouterHandler for the next IP packet by telling the RouterHandler to receive the next IP packet from the router. Then the JVARCore tries to find a matching observer for the IP packet or for the stream the IP packet belongs to. Eventually the packet gets passed to the found observer to display the payload in a readable form. Afterwards the JVARCore checks if the IP packet holds a task object and eventually passes it to the TaskCore for storage and scheduling after the task data has be extracted from the IP packet. Finally the JVARCore tries to find a matching FilterSet for the IP packet or for the corresponding stream. The matching FilterSet receives the IP packet to modify it. After the modification, the FilterSet returns zero to several new IP packets to the JVARCore. The JVARCore passes these received and eventually modified IP packets to the RouterHandler. The RouterHandler passes the IP packets to the Router, which routes them according to its routing table.

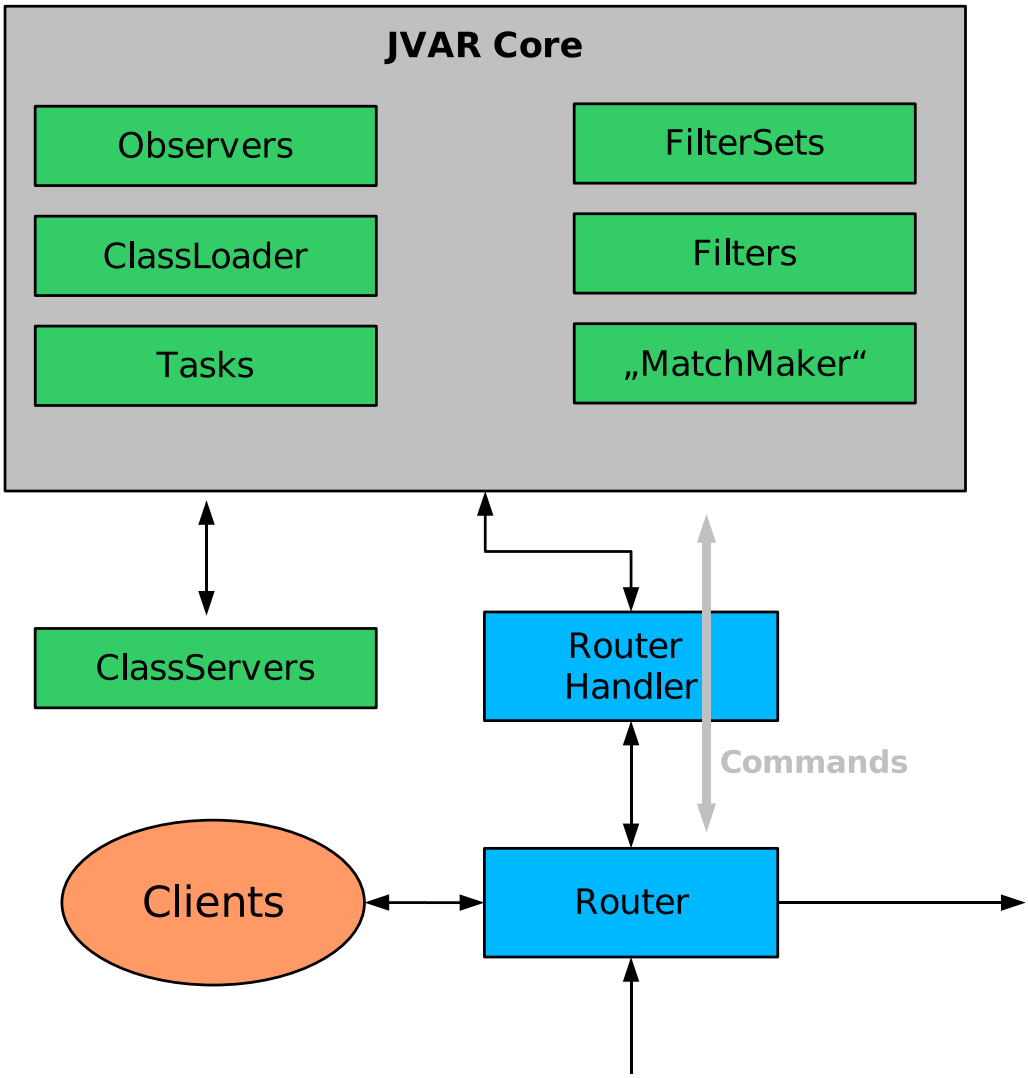

Figure 11: Basic setup of a node with JVAR

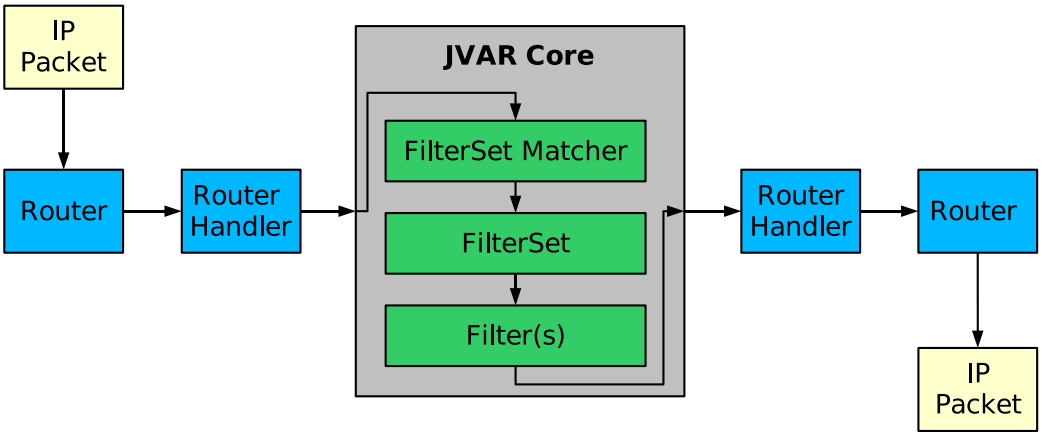

Figure 12: Packet flow with JVAR

2.5 JVAR compared to other active networking platforms

Often, active networking frameworks are limited either to use the in-band or the out-of-band approach (see also chapter 1). Both of these approaches have their advantages and disadvantages.

The in-band approach reacts very quickly and is very flexible. With each packet (or capsule) a new little program to manipulate itself inside an active networking node can be provided. There is though the limitation, that the size of the programs is limited (normally to 1 Kilobyte), so it can be transported with the packet. Also the overhead for a stream of packets of the same kind is high, since each packet could carry the same program to manipulate itself each time. The protocol used could be a proprietary one, which would not be widely distributed or only available for this special active networking implementation. Then it would be hard to integrate this new protocol into the real world and the existing routers.

The out-of-band approach has no real program size limitation. Programs can be sent in multiple packets and put together or downloaded from a remote location by the router. This action can be initiated by a packet holding all the necessary information. These programs and the packets holding the program information do not need to be implemented in a special protocol. Widely distributed protocols can be used to carry the information. All the manipulations can be done on packets that are using existing protocols (like IP, Ethernet, etc.). The reaction time though is very high. To apply changes, the whole download and initiating process has to be done first. Packets do not hold the programs directly. So if a situation changes, the corresponding programs to be loaded have to be triggered or instantiated first in the active node where the action has to be done.

A better solution is to take the best of both worlds, which also has been chosen for the JVAR framework. Programs and algorithms can be triggered to be downloaded from remote locations, as well as sent directly with some packets. All these programs then will work on certain streams, on which a FilterSet can be setup. FilterSets are the active networking components manipulating the packets for a certain stream. During the life cycle of a FilterSet, interactions between the packets and the FilterSets can happen. Additional configuration data or information can be embedded in the existing IP protocol, without modifying the protocol, just additional header information is added to the IP payload. Normal routers can then process the packets as if they were normal IP packets. The active networking nodes on the other hand can use the additional information to react on each packet. They do not have to wait for another trigger or program packet as with the out-of-band approach. Thereby the overhead, which would occur with the in-band approach by having each packet carrying a program, is reduced. But the flexibility of reacting on the spot by having additional configuration data or programs in a packet (if needed) is maintained.

Existing active networking frameworks [WGT98] written in Java often use special defined protocols for their active networking packets or capsules. This prevents them of being used easily in the real world with existing routers. The JVAR approach on the other hand uses the existing IP / UDP protocol, without having to re-program existing routers to support the active networking packets. The active networking data, programs or information is embedded within the IP packet, which allows every router to process them as normal IP packets. The active networking nodes on the other hand can extract and remove the additional data. When such a packet reaches the receiver, it is regarded and treated as a normal IP packet. Therefore all additional active networking manipulations and actions can be as well applied to any higher level protocol data or header information embedded inside the IP payload and of course also directly on the IP payload and headers.

2.6 Basic JVAR architecture issues and advantages

The basic advantages of the JVAR architecture include, that the framework is thread[5] free, well documented (javadoc), using a highly adaptive and configurable logging framework, doing efficient IP matching and routing list / table handling and that all vital classes have corresponding test classes using the JUnit [Bec02] framework (see also chapter 2.3 on page 23).

[5] Threads: quasi parallel executed actions

The whole framework is documented very well using the Javadoc standard. An automatically generated HTML documentation, which is completely linked, very easy to understand and allows browsing through the framework classes is provided. The source code in general has been documented very well. Someone using the framework or extending it can easily program new classes or build new active networking applications.

Threads can be hard to synchronise, and since only the examples use threads, the basic core of the JVAR framework is thread free. There are not any starving or unfair treated tasks or jobs handled by the framework. It is also very easy to follow a process happening in a node, since all actions are executed sequentially on one packet completely, before the next packet will be processed. Multiple JVAR instances running on the same machine cannot interfere or block each other through threads, since no deadlocks caused by threads can occur.

The logging framework [Gul03] used allows the user to configure the produced output of the JVAR framework and all classes involved. It is very easy to follow a packet stream and the actions happening on the payload data by following the output. This can be done either on the console or in a file, depending on the desired configuration. The logging framework has been built with the goal to have a very low resource usage, which makes the overhead minimal. For crucial applications, the logging can be set to show and deal only with severe errors.

The IP/UDP representing classes and all helper classes matching streams to certain actions, have been programmed to be very efficient and using low resources. All algorithms used have been either known to be fast or have been optimised for the given tasks. The access to the payload and the header fields of a packet have been simplified, so crucial actions can be done quick and easy.

By using a well known and widely used testing framework (JUnit), the quality of the framework can be augmented easily. On any extensions of the existing framework, the basic functionality can be verified to still work by using the provided test classes. The provided test classes can be extended or new ones can be added to assure the quality and stability of the framework and not break the whole system.

2.7 Packet modifiers: Filters and FilterSets

Filters

Filters do simple payload modifications. They can be concatenated to various kinds of chains[6]. They also have no access to the TaskCore of a JVARCore and cannot create any tasks (see also chapter 2.8 on page 32). Also no communication with the router via the RouterHandler can be done by Filters. They are the simple IP manipulating version and always belong to FilterSets, which often use multiple Filters for their actions.

To create a new filter, only one method has to be overridden after inheriting the AbstractFilter super class:

```
public abstract byte[][] manipulatePacket(byte[] aPacket);
```

A simple payload duplicating filter could look like this:

```
public byte[][] manipulatePacket(byte[] aPacket) {
  // duplicate packet
  byte[][] result = new byte[2][];
  result[0] = aPacket;
  result[1] = aPacket;
  logger.info("Duplicated packet successfully");
  return result;
}
```

FilterSets

FilterSets do the high level IP and JVAR packet modifications. They can use different concatenated / chained filters and have access to the TaskCore of a JVARCore. They can create tasks for delayed sending or for agent[7] and self notification (see also chapter 2.8 on page 32). Unlike filters, they can communicate directly with the router via the RouterHandler (see also chapter 2.11 on page 36) but they also can work directly on the payload of an IP packet without a filter.

To create a new FilterSet at least one method has to be overridden after inheriting the AbstractFilterSet super class:

```
public IP[] applyFilters(IP aPacket)
```

[6] Filters have references to the next filter in a chain and automatically pass the data to the next filter in this chain after they have modified it

[7] Agents in JVAR would be special FilterSets in remote JVARCores

To create a simple ”spy” FilterSet just some few simple lines of code are needed:

```
public IP[] applyFilters(IP aPacket) {
    //Simple "spy" (copy of IP packet + address rewrite)
    IP spyIP = new IP(aPacket.getIPPacket(), 0);
    spyIP.dest4 = 234;
    IP[] ipArr = {aPacket; spyIP};
    return ipArr;
}
```

2.8 The organisational units: JVARCore and Tasks

JVARCore

The heart of the framework is without any doubt the JVARCore. Main duties of a JVARCore include handling all traffic received by the router via the RouterHandler (see also chapter 2.11 on page 36). They receive, recognise, send and execute tasks. Each JVARCore holds a list of all instantiated FilterSets and a list active observers. These lists are stored as hash tables[8]. The observers are only interesting for simulation. JVARCores have to match IP packets to observers and FilterSets and pass modified packets back to the router via the RouterHandler.

The "MatchMaker" is the main method doing the matching of IP packets to FilterSets, Observers or to the TaskCore. The FilterSet selection is done FCFS (first come first serve) and therefore only one FilterSet matches an IP packet. The matching FilterSet can return more than just one IP packet. Other matching objects can be inserted in the match making process (the framework can be easily extended). The match making process is completely thread free. The JVARCore has free resources for the next packet, after it handled the current IP packet in the process completely.

Tasks

Tasks are being handled by the JVARCore via the TaskCore, which deals with the scheduling of the tasks. Tasks are the main active triggers in the network. A user or a JVARCore sets up the active components (FilterSets) by tasks. This can be done even for remote locations. Tasks have different kind of functions (for example remove / add FilterSets on certain flows). JVARCores and FilterSets can communicate with other foreign instances of themselves via tasks (for example

[8]Hash tables store key and value pairs, which can be found / accessed by their keys

agent communication to get information about foreign resource usage). IP packets can be deferred with the help of tasks for delayed transmission. Actions can be triggered by tasks. Tasks can be extended for other applications very easily. Tasks can be executed at a certain moment (a positive delay from now) and are executed once, several or even infinitely.

Different kinds of tasks exist, which will be explained now more in detail.

TASK_FILTERSET_ADD (= integer 0) tells the JVARCore to instantiate a FilterSet with certain parameters and then to add it to its hashtable storing them. From that moment on, this FilterSet is active on the defined IP packet stream.

TASK_FILTERSET_REMOVE (= integer 1) tells the JVARCore to remove a certain FilterSet from duty. It gets removed from the hashtable and is no longer working on its previously defined IP packet stream.

TASK_NOTIFY (= integer 2) notifies a sleeping FilterSet or other sleeping objects to wake up and to continue its actions.

TASK_LOADCLASS (= integer 3) triggers a class loading mechanism. The class will get loaded by a class loader in the JVARCore. The class loader knows how to deal with the information this special task provides and how the class can be loaded.

TASK_SENDIPPACKET (= integer 4) schedules an IP packet for deferred transmission. The IP packet gets put into the TaskCore and will be sent to its receiver via the JVARCore / RouterHandler / router at the predefined moment. Also FilterSets in other JVARCores can be periodically informed about some current status information of the JVARCore or the network itself.

Finally a TASK_DUMMY (= integer 5) exists, which does not really do any real action, but can be used for keeping the JVARCore alive or to test different timing issues.

Tasks have various parameters, which hold objects, that have different kind of meanings depending on the task type.

TASK_NOTIFY:
taskObject1 = object: reference to object being notified
taskObject2 = nothing

TASK_DUMMY:
taskObject1 = nothing
taskObject2 = nothing

TASK_FILTERSET_ADD:
taskObject1 = string: name of filterset class
taskObject2 = ComplexIPMask: complex IP mask for the filterset to match

TASK_FILTERSET_REMOVE:
taskObject1 = ComplexIPMask: complex IP mask of the filterset to remove
taskObject2 = nothing

TASK_LOADCLASS:
taskObject1 = string: name / address of class providing server
taskObject2 = string: name of the class to be loaded

TASK_SENDIPPACKET:
taskObject1 = IP: IP packet that has to be sent
taskObject2 = nothing

The following example of a simple task loads a filter from a ClassServer. The task will be executed immediately, since the first parameter (the delay in milliseconds) is zero. It will be executed only one time, because the second parameter is set with the number one. Then follows the task type and the special object parameters described before:

```
Task taskLoadFilter =
  new Task(0, 1, Task.TASK_LOADCLASS,
           "localhost:8888",
           "com.brogle.jvar.filters.SimpleFilter");

// add the serialised task to an IP payload
IP ipTaskLoadFilter =
  IPFactory.getIP(192,168,0,1,192,168,10,1,222,0,
                  taskLoadFilter.getSerializedObject());

// route the IP packet via a RouterHandler (rh0)
rh0.routeIPPacket(ipTaskLoadFilter);
```

To distinguish IP packets holding tasks from normal IP packets, the protocol number in the IP header of a task IP packet is set to 222. Tasks are filtered directly

by the JVARCore before any FilterSets are matched to the IP packet. An alternative would be to create a distinguished FilterSet, which handles tasks by getting them from the IP flow and adding them to the TaskCore. Only JVARCores and FilterSets have access to the TaskCore and only they are allowed to create tasks.

2.9 Watching the action: Observers and Clients

Observers

Observers can be connected to JVARCores and clients. With the help of an observer, the payload of an IP packet (matching a ComplexIPMask) can be analysed. With the help of observers, for example a MJPEG stream and the way it is modified in certain nodes can be observed. At the moment two implementations of observers and one common super class for them exist.

All observers have to inherit from the **AbstractObserver**. The AbstractObserver provides a common access interface to easily connect JVARCores and clients to the specialised observers.

An **ImageObserver** is used to analyse how UDP JPEG payload gets modified on certain JVAR nodes. The JPEG pictures are shown in window displaying the image and different image information is printed to the console. The **PayloadObserver** shows the payload of an IP / UDP packet. The text / payload is showed in a windows and also the same text / content is printed to the console.

The observers know by passing options on creation if they have to deal with additional special JVAR header information. In that case they show the payload without this additional JVAR header information.

To create a new Observer only one method has to be overwritten after it inherits the abstract superclass AbstractObserver:

```
public abstract void show(IP ip);
```

Simulated clients

The client written in Java, which is provided with the JVAR framework allows to analyse the packets it receives through an attached observer. Clients are directly connected to the already mentioned routers written in Java (see also chapter 2.11 on page 36). Like the router the client has a lightweight implementation, which makes it easy to understand how clients are used and programmed.

2.10 Class loading and serving

SocketClassLoader

Class loaders load classes that are requested by remote JVARCores or users. Since different class providing servers or services can exist, different class loaders would have to be implemented. With the JVAR framework a SocketClassLoader is provided, which loads classes from a SocketClassServer.

An abstract super class called the AbstractClassLoader is provided, to have a common interface for the JVARCore to be able to use any kind of class loader. The real implementation of a class loader just has to inherit from the AbstractClassLoader. The newly implemented class loader also has to provide a customised constructor and a class loading method.

SocketClassServer

The implementation of the SocketClassServer or any other class server does not have to inherit from any abstract super class, since different kind of services could be provided. Only the corresponding class loader has to be adapted to the specific class server it has to connect to.

The example SocketClassServer provided offers classes to be loaded via a socket on a specific port.

2.11 Routing through the network: Routers / RouterHandlers

RouterHandlers

RouterHandlers are the communication interface between a JVARCore and different kinds of routers. To connect a JVARCore to a router successfully, the router has to provide some basic communication capabilities, which then are handled by a specific RouterHandler for this router. Since the RouterHandler inherits a basic interface from the abstract super class AbstractRouterHandler, the JVARCore can access the router in the same way it would access any other router. Thus, the JVARCore implementation and the whole JVAR framework is completely router independent. Any kind of router can be used as long as the router provides the mentioned basic interface to access itself.

For the examples and the framework, a router written in Java exists to emulate a network with interconnections, which can be setup on one single computer. The corresponding RouterHandler is also provided.

Fragmentation and defragmentation of IP packets have to be done in the RouterHandler that has to be implemented for a specific router. All commands that are sent by the method `sendRouterCommand` can use any "still to be defined" protocol, which then also has to be implemented in the JVARCore.

The `setupFilter` and `removeFilter` provided functionality, which would allow to tell a router that the JVARCore connected to it is only interested in a certain stream of the flow going through a router, that has not been used so far. The provided demo router written in Java does not provide this special functionality, since it is only used for simulation. These two methods are intended as basic interface methods any other RouterHandler should implement. Therefore the JVARCore deals only with packets it really needs to get for the its setup Filters and FilterSets. Unnecessary matching actions inside a JVARCore would be avoided. The matching to distinguish packets which would have to be handled by the JVARCore and the ones just passing through without any modifications would directly be done by the router. This procedure would keep the resource usage of the JVARCore low if there are not any packets (passing through the router) in which the JVARCore would be interested.

Emulated routers

The router written in Java, which is provided with the JVAR framework to emulate several routers [BBKW] on one computer has some special features. Routers do not use threads, which makes them easy and straight forward to use. This also helps to simulate an active networking application in a predictable serial manner. The examples provided with the JVAR framework always trigger the different routers serially. Thus it is easy to compare the behaviour of an example while executed with the expected behaviour.

Routers can have simulated clients connected to themselves. They are simply coded, which helps understanding the router implementation easily. Routers are connectible to other routers and they use the helper classes (for example IPAdr) provided with the JVAR framework for their routing tables.

2.12 Different packets: IP and JVAR and matching them

All implemented protocol and packet classes inherit from the abstract "Packet" class. This class provides basic byte and array handling.

IP

An instance of the IP class represents one IP packet. This can be either an IP or an UDP packet. The basic functions to handle IP addresses and the UDP settings are implemented. The class has been optimised on the low level access functions. The header fields of an IP packet can be set and read directly.

The calculation of the checksum field has not been implemented, since this should be done in the router itself to increase overall performance of the JVAR framework. If the router does not provide this functionality, the checksum can be calculated in the corresponding RouterHandler.

JVAR

The JVAR protocol / packet class allows the framework to add some extra information to existing IP classes. The additional header information is placed at the beginning of the IP payload field in an IP packet. For JVAR the header definition see also figure 13 on page 39.

The following special actions on JVAR packets realised with special FilterSets are implemented:

Explicit multicast: The JVAR header option field holds multiple IP addresses as destination addresses for the IP packet. In each JVARCore where a JVAR packet of this type travels through and is matching a flow rule configured in the JVARCore, the links of the corresponding router get analysed. If some of the destination addresses will be reached by different links in the router, the FilterSet multiplies the JVAR packet. Then the option fields get adapted to the different links and the destination addresses for the newly created JVAR packets get rewritten. Finally the packets get sent through the RouterHandler to the router, which routes them according its routing table.

Explicit route: The JVAR header option field holds multiple IP addresses as designated travel points that have to be visited. In each JVARCore where a JVAR packet of this type travels through and is matching a flow rule configured in the JVARCore, a check is being made if the packet is at the next defined point in the visit list. If matching a point if this list, the destination address of the IP packet gets replaced with the next target in the list and the same address gets removed from list in the option field.

For an example of a concrete JVAR header, see 14 on page 39.

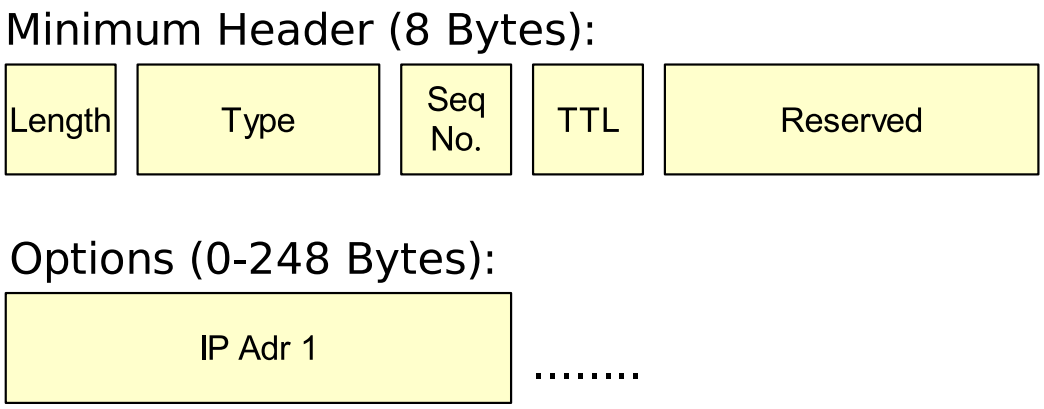

Figure 13: JVAR header definition

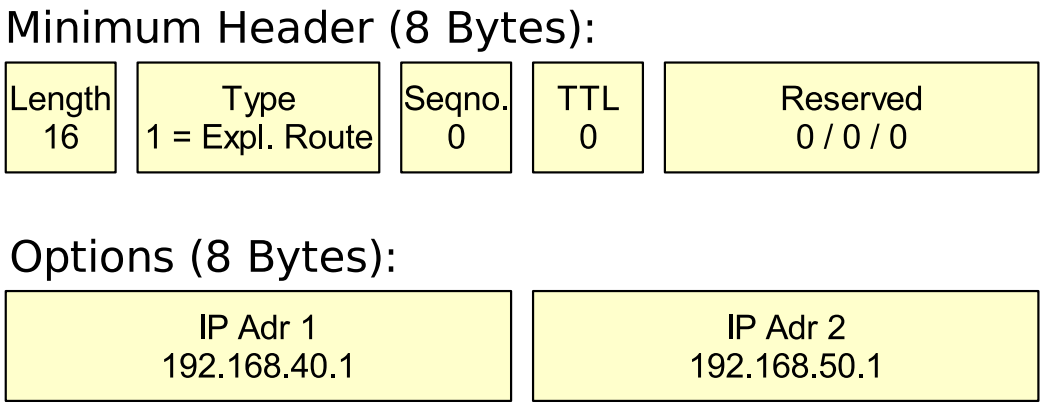

Figure 14: JVAR header example

Packet matching

A ComplexIPMask compares and checks different parts of an IP / UDP packet. Comparison is made on the source address, the destination address, the protocol type, the TOS (Type of Service) field. If it an UDP packet, the the UDP source port and the UDP destination port get compared as well. For all those parameters, the value `-1` is regarded as a wild-card, which will always match.

Because it implements the `java.io.Serializable`[9] interface, the ComplexIPMask can be serialised and sent in an IP packet to setup a task using such a ComplexIPMask, for example for FilterSet setup.

There are 2 main constructors provided. Either a custom ComplexIPMask is created setting all the mentioned IP matching parts manually:

```
public ComplexIPMask(IPAdr source, IPAdr dest,
                     int protocol, int tos,
                     int udpsrc, int udpdst)
```

Or all necessary information is extracted directly from an existing IP packet:

```
public ComplexIPMask(IP ip)
```

The check itself is done very efficiently and aborts all further matching tries on the first mismatch, which leads to a better performance.

2.13 Security and logging

Security and safety issues

Safety and security [Bro01] are very important issues when it comes to active networking. It has to be assured that only allowed modifications are applied. Also mechanisms to ensure that the program senders or trigger sources are trustworthy could be introduced. Active networks have the ability to allow users to add their programs to the network nodes. This way, they can modify or configure the network to their wishes and they can provide new services. Giving this ability to a user increases safety and security problems. Distinctions between safety and security should be made. A security framework protects an active node from malicious actions. A safety framework protects it from trusted parties that unintentionally take dangerous actions. Attacks, to which an active node is exposed to, are greater than for the passive networks. Possible security problems that could arise are the misuse of active node programs by other running programs on the active node,

[9] java.io.Serializable: interface providing basic object serialisation and deserialisation methods

the mis-usage of an active node with programs, the abuse of programs by an active network node or the malpractice of active programs through the infrastructure laying beneath.

Implemented security

To cover some security issues, the following security procedures have been implemented in the JVAR framework. To create a random key to encrypt the payload of an IP packet, the secure random generation provided with the SUN Java J2SE (Java 2 Platform Standard Edition) version 1.4 has been used. This algorithm is based on the IEEE P1363 standard for creating safe random numbers. SUN calls the algorithm “Random Number Generation (RNG) Algorithms SHA1PRNG“. The implementation follows the IEEE P1363 standard, Appendix G.7: "Expansion of source bits" and uses SHA1 as the foundation of the PRNG (Programmable Random Number Generation). It computes the SHA1 hash over a true-random seed value active with a 64-bit counter which is incremented by 1 for each operation. From the 160-bit SHA1 output, only 64 bits are used. For more information about the algorithms and the methods used as well about the safety of the whole procedure, see also [IEE97].

Logging

To catch the runtime information the classes produce while running on a JVAR node setup, a logging framework [Gul03] has been used. This logging framework can be easily configured and turned off. It has been implemented to be very resource effective and not time consuming. Almost all classes in the JVAR framework create some logging output. Since there are different kind of logging levels, which denote the importance or severity of a message, the logging framework can be configured to show only messages of a certain importance or only messages from certain classes.

The output of the logging messages can be redirected to different kind of files, to streams or even to the log facilities[10] the underlying operating system provides. Different kind of logging levels can be redirected to different channels. Some files register the normal information while others hold all fatal errors happening in the JVAR node.

For more details on the logging framework, see also appendix C on page 88.

[10] Logging facilities like the syslog on Linux or the event handling on Windows

3 Evaluation

For the presented example active networking applications, there will be first a short overview and then details about the implementation presented.

The overview will explain the motivation behind the example applications and the problems involved. The solution using the JVAR framework and some possible alternatives solving the problem without the use of the JVAR framework will be shown as well.

The explanation of the implementation will consist of different parts. The basic IP / UDP / port configuration and the used JVAR protocol extensions (if needed) will be explained. The description of the behaviour of the example and an example image of the network layout or of the actions during the execution of the example will be given as well. Finally the chosen approach of active networking (see chapter 1.2 on page 1) will be explained and the factors influencing the choice of the the in-band, out-of-band or a mixed form of both approaches will be described.

All these applications have been programmed and tested. For the different concrete implementations see also the demos provided with the JVAR framework. All demos produce extensive logs, which allow the user to see what is happening inside the JVARCores and all other involved classes.

See also appendix C on page 88 for information about how to configure the logging framework. All actions are displayed in the attached observers, which will show the text or image information being processed in the attached JVARCore.

3.1 Simple network layout example

Overview

This example shows the basic network layout, which has been used for the simple examples provided. It consists of a simple private class network (192.168.x.x). The network consists of 6 virtual routers and one simulated client attached to the last router at the end of the network. The simulated routers and simulated clients used, are the ones provided with the JVAR framework. The network layout and interconnections are setup by the BasicDemoSetup class which creates the different instances of the "Router" and "Client" classes and connects them according to the desired layout. The default route passes via the routers two and three (192.168.20.1 and 192.168.30.1). See also figure 15 on page 44.

Implementation

Each JVARCore loads remotely a SimpleFilterSet and a SimpleFilter. This loading gets triggered at the start of the whole example, where the IP packets with the class loading commands get injected into the example network through the first router. The matching flow rules in the JVARCores, which will trigger the use of the FilterSets, are configured to match exactly the source / destination addresses and the UDP protocol (port 17) of the example IP packets, which are injected into the virtual network setup. Each JVARCore on the route modifies the IP packet. A simple Filter and a simple FilterSet modify the IP packets. The packets only pass through the Filter and the FilterSet without actually being modified. In the log files, the console output and on the screen, the actions and content of the IP packet are displayed. This example mainly shows how the JVAR framework can be used and how a simple setup with some simple rules and simple Filters / FilterSets can be realised. The out-of-band approach is used to realise this example since all Filters and FilterSets are loaded at the startup of the example, triggered by the SimpleDemo class, before any normal network traffic occurs.

Looking at the log output in table 1 on page 44 shows, that a packet containing a task gets recognised, and afterwards gets added to the TaskCore. From the TaskCore the task gets executed by the JVARCore, requesting the class to be loaded from a SocketClassServer. The SocketClassServer then hands the requested class to the JVARCore.

The log output in table 2 on page 45 shows the processing of an example packet. The packet is destined for a specific client and travels through the network and through the routers with the attached JVARCores. The observers and FilterSets are matched to the stream of the packet and actions are taken accordingly. Finally it reaches the client which then also displays the message (payload of the packet) it received to the log and console.

3.2 Complex network layout example

Overview

The network setup for this example uses like the example presented in chapter 3.1 on page 42 virtual routers and simulated clients provided with the JVAR framework. The class ComplexDemoSetup creates the different instances of the "Router" and "Client" classes and connects them according to the desired layout. The example network layout consists of five private class subnets 10.10.x.x / 10.20.x.x / 10.30.x.x / 10.40.x.x / 10.50.x.x. The default route through the net-

```
SocketClassServer - Created server socket on port: 8012
Router - Routing packet 192.168.10.1
Router - Found matching router 0.0.0.0/0.0.0.0
JVARCore - JVARCore1: no observer found
JVARCore - JVARCore1: Found a matching task
TaskCore - Task found queue ready for execution
SocketClassServer - Initialised listener on socket server
SocketClassServer - Received request for className:com.[...].SimpleFilter
SocketClassServer - Found and loaded class:com.[...].SimpleFilter
SocketClassServer - Sent class:com.brogle.jvar.filters.SimpleFilter
JVARCore - JVARCore1: Executed TASK_LOADCLASS
JVARCore - JVARCore1: no observer found
JVARCore - JVARCore1: Found a matching task
TaskCore - Task found queue ready for execution
SocketClassServer - Initialised listener on socket server
SocketClassServer - Received request for className:com.[...].SimpleFilterSet
SocketClassServer - Found and loaded class:com.[...].SimpleFilterSet
SocketClassServer - Sent class:com.[...].SimpleFilterSet
JVARCore - JVARCore1: Executed TASK_LOADCLASS
JVARCore - JVARCore1: no observer found
JVARCore - JVARCore1: Found a matching task
TaskCore - Task found queue ready for execution
JVARCore - JVARCore1: Executed TASK_FILTERSET_ADD with com.[...].SimpleFilterSet
JVARCore - JVARCore1: no observer found
JVARCore - JVARCore1: no filterset found
```

Table 1: Log output showing class loading

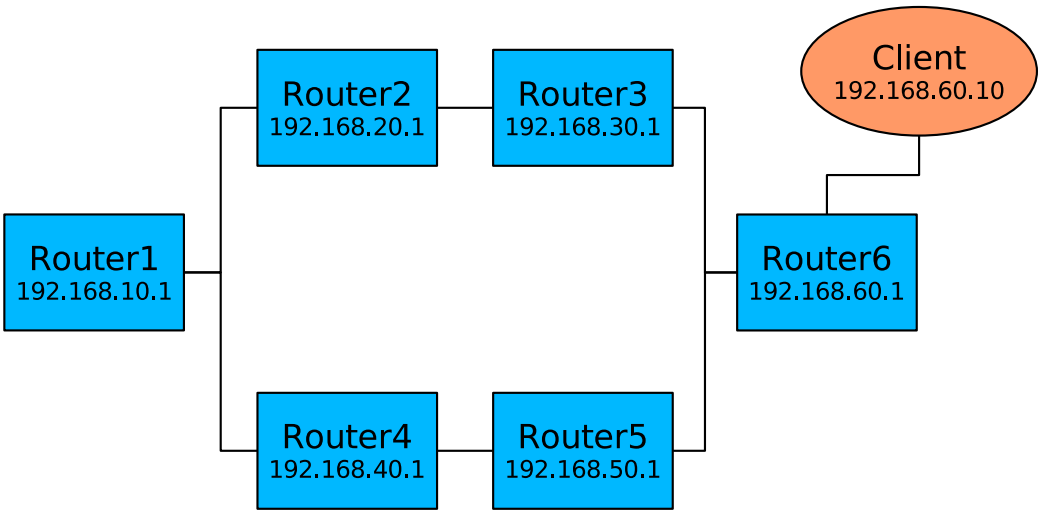

Figure 15: Simple network layout

```
192.168.10.1 Router: Packet for 192.168.60.10 - This is a Test - This is a Test
JVARCore - found an observer
JVARCore - JVARCore1: found a filterset
Router - Routing packet 192.168.60.10
Router - Found matching router 0.0.0.0/0.0.0.0
192.168.20.1 Router: Packet for 192.168.60.10 - This is a Test - This is a Test
JVARCore - found an observer
JVARCore - JVARCore2: found a filterset
Router - Routing packet 192.168.60.10
Router - Found matching router 0.0.0.0/0.0.0.0
192.168.30.1 Router: Packet for 192.168.60.10 - This is a Test - This is a Test
JVARCore - found an observer
JVARCore - JVARCore3: found a filterset
Router - Routing packet 192.168.60.10
Router - Found matching router 0.0.0.0/0.0.0.0
192.168.60.1 Router: Packet for 192.168.60.10 - This is a Test - This is a Test
JVARCore - found an observer
JVARCore - JVARCore6: no filterset found
Router - Routing packet 192.168.60.10
.Router - Found matching client 192.168.60.10/255.255.255.255
192.168.60.10 Client: Packet for 192.168.60.10 - This is a Test - This is a Test
```

Table 2: Log output showing packet handling

work from left to right passes via the 10.30.x.x subnet. There is an interconnection between the 10.20.x.x and the 10.30.x.x subnets. There are five clients in the network, where two of them are in the 10.20.x.x subnet and one is in each of the 10.30.x.x / 10.40.x.x / 10.50.x.x subnets. See also figure 16 on page 46.

Implementation

To test the example network setup, a packet containing some simple text payload is sent to each of the clients. No Filters and FilterSets are being loaded (by the class loader) at the startup of the example. No Filters and FilterSets are being instantiated during this example. This example is used to show how to setup a more complex layout of a network to be simulated with the JVAR framework. Since no Filters or FilterSets are being used, this example uses neither the in-band nor the out-of-band approach.

Figure 17 on page 46 shows how the packets for the different clients are sent trough the network and how the different clients receive these packets. The observer for JVARCore attached to the router 10.10.112.1 shows all packets being sent. The other observers attached to the clients show how the right packets get received by the clients.

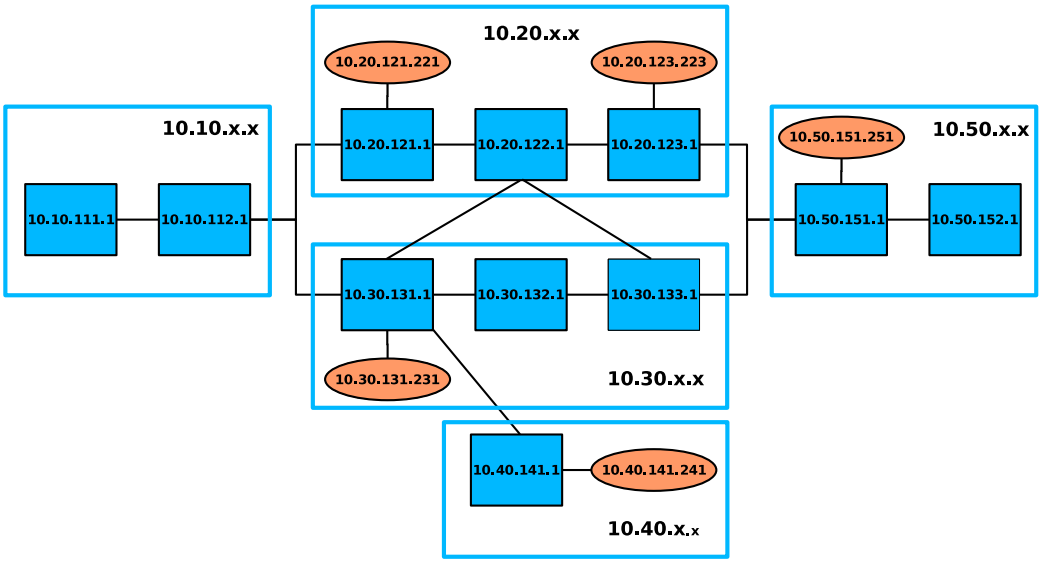

Figure 16: Complex network layout

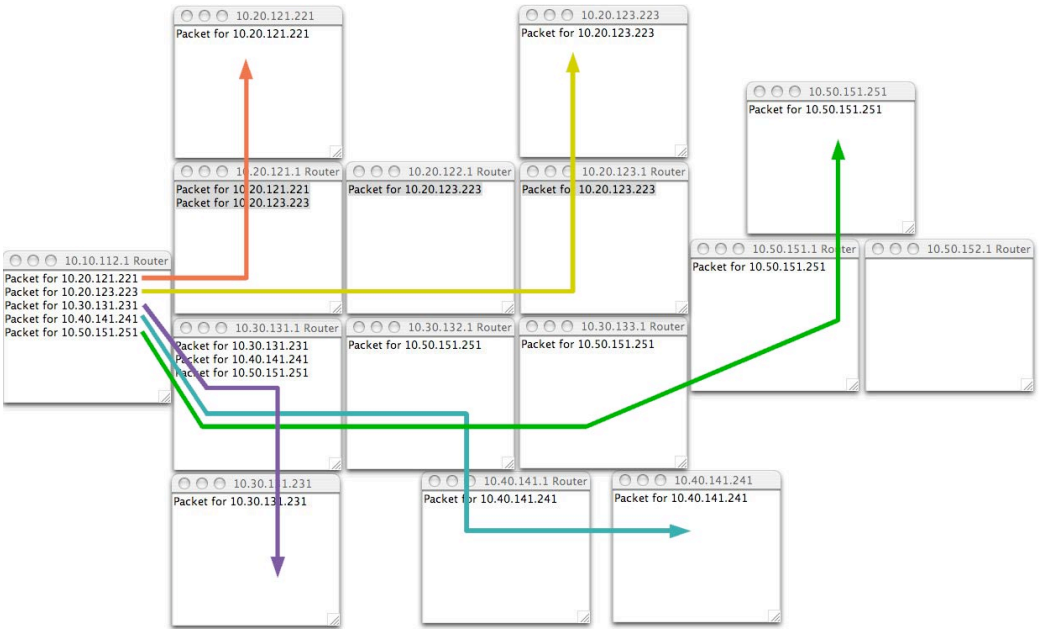

Figure 17: Screen-shot of complex setup example

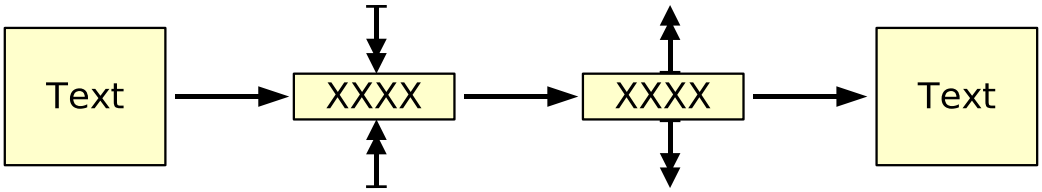

Figure 18: GZIP packing of payload

3.3 GZIP tunnel

Overview

A network provider wants to optimise bandwidth usage or wants to get some additional reserved resources in a subnet. From this moment there would be less bandwidth in the affected subnets that could be used for the normal services. To solve the problem with JVAR, a GZIP or other tunnels in and through selected subnets could be created. Then no bandwidth reduction for the general services would be needed, since the additional reserved bandwidth for the special services could be provided by compressing the normal traffic. Alternatives without using JVAR could include the usage of a packet shaper, which would reduce the overall granted bandwidth for the normal services. This would keep some bandwidth reserved for the special services.

Implementation

In this example the network layout is the same as explained in section 3.1 on page 42. Text in the UPD payload part of an IP packet (no special protocol needed) will be sent through the example network. In a certain part of the network the payload gets sent through a GZIP tunnel, which will compress the payload of a packet as long as it is in this tunnel. There is one compress point and one decompress point to setup a GZIP tunnel. The FilterSets in these points have strict rules. They match only on the specific source and destination addresses and the UDP protocol of the example packets sent through the network. All the actions and modifications made are completely transparent to the users (client and sender). They do not know, that their packets have been compressed and decompressed on the way through the network. The GZIP tunnel uses the GZIP algorithm, which is included in the Java libraries. See also figure 18 on page 47.

The log output in table 3 on page 48 demonstrates the setup of the compression and decompression filters and FilterSets in the corresponding JVARCores after they have been loaded remotely from a SocketClassServer. This example uses

```
JVARCore - JVARCore1: Found a matching task
TaskCore - Task found queue ready for execution
[...]
SocketClassServer - Sent class:com.brogle.jvar.filters.GZIPOutputFilter
JVARCore - JVARCore1: Executed TASK_LOADCLASS
[...]
TaskCore - Task found queue ready for execution
JVARCore - JVARCore1: Executed TASK_FILTERSET_ADD with [...]GZIPOutputFilterSet
[...]
JVARCore - JVARCore3: Found a matching task
TaskCore - Task found queue ready for execution
[...]
SocketClassServer - Sent class:com.brogle.jvar.filtersets.GZIPInputFilterSet
JVARCore - JVARCore3: Executed TASK_LOADCLASS
[...]
TaskCore - Task found queue ready for execution
JVARCore - JVARCore3: Executed TASK_FILTERSET_ADD with [...]GZIPInputFilterSet
```

Table 3: Log output showing setup of (de)compression point

only the out-of-band approach since all Filter and FilterSet setups occur before the normal network traffic starts and the packets do not modify these Filters and FilterSets anymore.

The GZIP demo log output in table 4 on page 49 shows, that the payload of the IP packet gets compressed at the JVARCore attached to the second router. At the JVARCore attached to the third router the payload gets decompressed. Only the sending client attached to the first router and the receiving client attached to the last router send and get the message in an uncompressed form. On its way through the network the IP packet is passed through the GZIP tunnel. The quantitative benefit gained by the compression depends on the nature of the payload. An Apache (web server) log file can be compressed by the factor 1:20 to 0.05% of its original size. The text of this diploma work can be compressed by the factor 1:4. A GIF / JPEG image or already compressed data cannot be reduced in size anymore and typically its size even grows when a compression is reapplied.

3.4 JPEG stream manipulation

Overview

A user wants to send (M)JPEG images or streams to another user. The overall network traffic could be high or the connections of the users could be very slow. This could lead to packet drops, timeouts and asynchronous flows, which would

```
192.168.10.1 Router: Packet for 192.168.60.10 - This is a Test - This is a Test
JVARCore - found an observer
GZIPOutputFilter - Wrote data to GZIPOutputStream successfully
JVARCore - JVARCore1: found a filterset
Router - Routing packet 192.168.60.10
Router - Found matching router 0.0.0.0/0.0.0.0
192.168.20.1 Router: ?HL?N-QH?/R0?4?34??33?34P???,V?D????1?*:
JVARCore - found an observer
JVARCore - JVARCore2: no filterset found
Router - Routing packet 192.168.60.10
Router - Found matching router 0.0.0.0/0.0.0.0
192.168.30.1 Router: ?HL?N-QH?/R0?4?34??33?34P???,V?D????1?*:
JVARCore - found an observer
GZIPInputFilter - Read data from GZIPInputStream successfully
JVARCore - JVARCore3: found a filterset
Router - Routing packet 192.168.60.10
Router - Found matching router 0.0.0.0/0.0.0.0
192.168.60.1 Router: Packet for 192.168.60.10 - This is a Test - This is a Test
JVARCore - found an observer
JVARCore - JVARCore6: no filterset found
Router - Routing packet 192.168.60.10
Router - Found matching client 192.168.60.10/255.255.255.255
192.168.60.10 Client: Packet for 192.168.60.10 - This is a Test - This is a Test
```

Table 4: Log output showing GZIP compression / decompression

then result in incomplete or slow streams. With JVAR, the image quality could be changed on certain active nodes. This modification would depend on the overall bandwidth usage of the network or on the free resources available at the very moment the packet (image) passes through such a node. Alternatives without JVAR could include a client side (at the users) negotiation of available bandwidth. The processing of the images would happen according to the negotiated and other informations at the sender. However, the sender could not react that fast to resource changes in the network. The network would have to be analysed periodically and new negotiations between sender and receiver would have to be done before the modification method could be changed.

Implementation

The network layout for this example is the same as explained in section 3.1 on page 42. A JPEG image is sent in the UDP payload of an IP packet (no special protocol needed). This payload gets modified at different JVARCores by strict filters, which are matching on the specific source and destination address with the UDP protocol of the example IP packets. These filters modify the image with certain actions, which include size reduction, bigger compression and black and

Figure 19: Modifying image payload

white conversion. These actions are completely transparent to the users (recipient and sender). All packets will be delivered and the overall frame rate is guaranteed. No packets will be dropped, only the quality of the image gets adapted to the current situation of the network usage. Therefore the delivery can be granted with a certain maximum delay produced by the image manipulation. The example uses the libraries integrated in Java for the JPEG image manipulation. The images get reduced in size or colour depth (conversion to black and white). See also figure 19 on page 50.

Figure 20 on page 51 shows the final state of the example after its execution with observers attached to the routers and the client. The colour conversion after the first router (192.168.20.1) is shown. Also the dimension resize of the picture before it reaches the client (192.168.60.10) via the last router (192.168.60.1) is recognisable. As explained in section 3.8 on page 57 the image manipulation on the benchmarking test system took about 61 ms. This example uses only the out-of-band approach, since the image modifying nodes are setup before the regular network traffic starts and no further packet manipulating setup information is passed to the nodes during the image transfer.

3.5 Explicit routing

Overview

A network provider wants to optimise the bandwidth usage and distribute the traffic over the subnets evenly. Also a sender could want to use only fast routes. These wishes would originate from the bad utilisation of the free network resources. While some subnets or network pathways would be heavily used, others would be

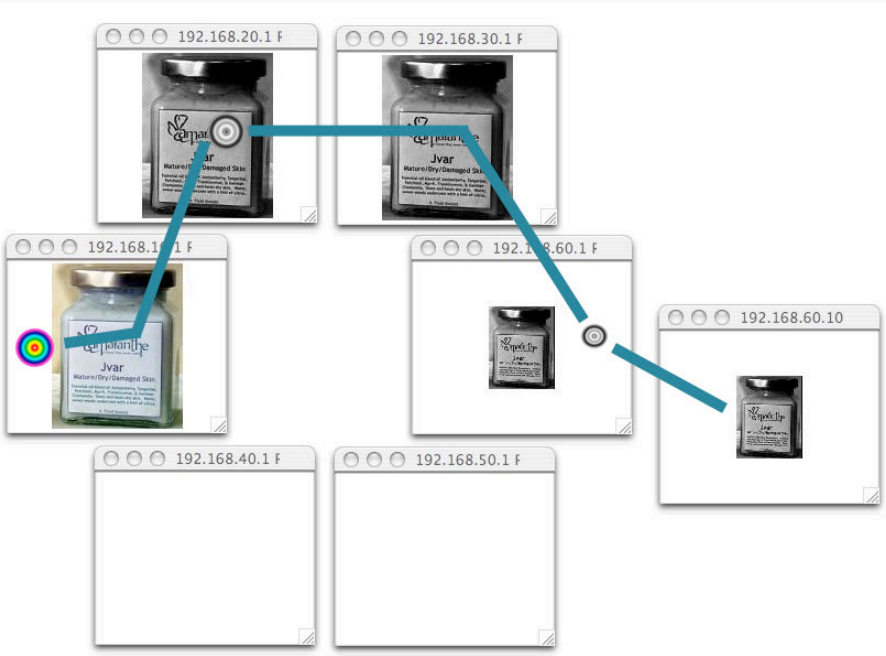

Figure 20: Screen-shot of JPEG stream manipulation example

only slightly used or completely unused. With JVAR, packets can be sent explicitly via a specified route. The nodes and packets themselves can decide where the packets have to go through. Alternatives without JVAR would be to rewrite periodically the routing tables in the routers. This procedure would react less faster on changing situations. There would be some overhead and performance issues to consider when these routing tables would have to be rewritten.

Implementation

The network layout for this example is the same as explained in section 3.2 on page 43. For this example a special JVAR protocol is used. The option field of the JVAR header holds different IP addresses that have to be visited. If these visiting points are reached, the destination address in the IP header of a packet matches with the address of the local router. Then the matching FilterSet modifies the packet and removes the first address in the JVAR header option field of the packet. The removed IP address from the JVAR header option field is then used as the new destination address for the packet (rewrite of destination address). The packet then gets sent back on its way through the network. See also figure 21 on page 52.

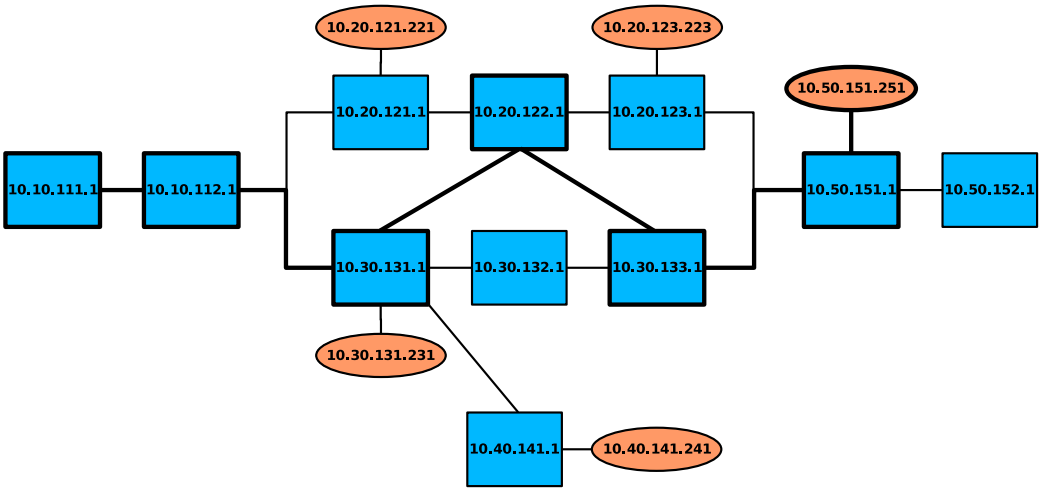

Figure 21: Explicit route setup

The log output on table 5 on page 53 and the figure 22 on page 53 show, that the packet does not use the default route that would go via the 10.30.132.1 router. Instead it gets redirected via the 10.20.122.1 router and then the packet goes back on the default route. The packet address gets rewritten on each JVARCore as mentioned before. This example uses both the in-band and out-of-band approach, since first the nodes get setup for a certain stream (out-of-band) and then the packets themselves (in-band) provide the information of where and how to route them.

3.6 SplitCrypt tunnel

Overview

A user wants to send to another user sensitive data. They do not trust the networks between them [GBBb]. Therefore any third party could access the sensitive payload sent. To solve this problem with JVAR, a special 2-way split point can be created. There the payload gets encrypted by a key. The encrypted payload and the key get sent on 2 different routes (= 2 way encrypted tunnel). Alternatives not including JVAR could use a public / private key signing [RSA77] on clients.

Implementation

The network layout for this example is the same as explained in section 3.2 on page 43. An encryption tunnel, which encrypts the UDP payload of an IP packet (no special protocol) will be created. One encrypt point, which splits the packet into two new packets is introduced to the network. Also one decrypt point, which

```
[...]
JVARCore - JVARCore 112: found a filterset
Router - Routing packet 10.10.112.1
Router - Found matching router 0.0.0.0/0.0.0.0
10.30.131.1 Router: Packet for 10.50.151.251
JVARCore - found an observer
JVARCore - JVARCore 131: found a filterset
Router - Routing packet 10.20.122.1
Router - Found matching router 10.20.0.0/255.255.0.0
10.20.122.1 Router: Packet for 10.50.151.251
JVARCore - found an observer
JVARCore - JVARCore 122: found a filterset
Router - Routing packet 10.30.133.1
Router - Found matching router 10.30.0.0/255.255.0.0
10.30.133.1 Router: Packet for 10.50.151.251
JVARCore - found an observer
JVARCore - JVARCore 133: found a filterset
Router - Routing packet 10.50.151.1
Router - Found matching router 0.0.0.0/0.0.0.0
10.50.151.1 Router: Packet for 10.50.151.251
JVARCore - found an observer
JVARCore - JVARCore 151: found a filterset
Router - Routing packet 10.50.151.251
Router - Found matching client 10.50.151.251/255.255.255.255
10.50.151.251 Client: Packet for 10.50.151.251
```

Table 5: Log output showing explicit route packet flow

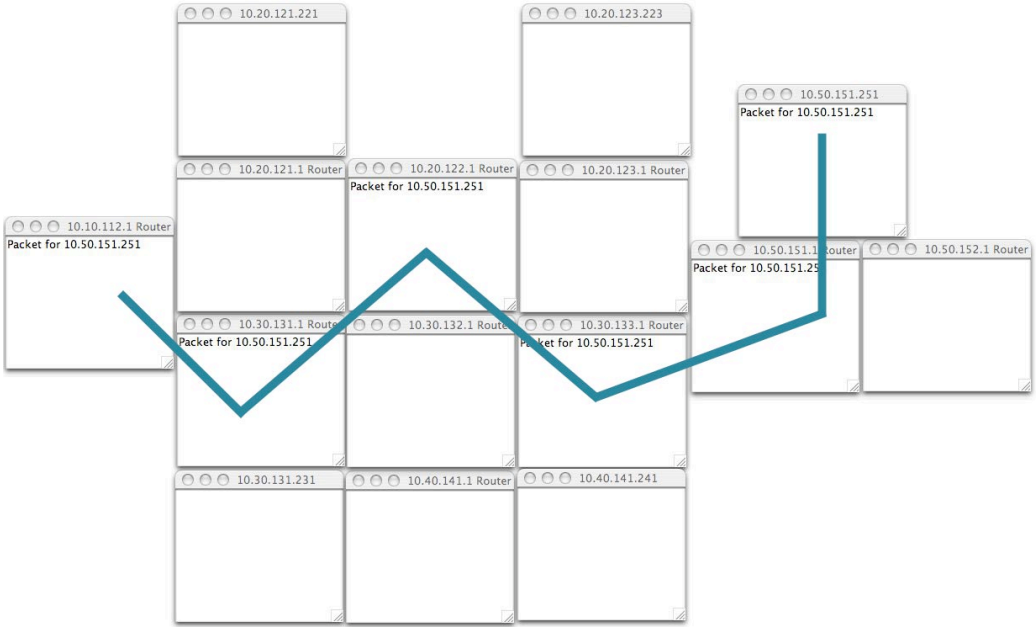

Figure 22: Screen-shot of explicit route example

restores the original data from the two packets mentioned is added to the network. These two points create the encrypted tunnel. The payload of the original IP packet is in plain text. Therefore the encryption and decryption can be followed with the observers and with the output the example generates. The random key, which will encrypt the data at the encrypt point has the same size as the data payload. This key is generated at the encrypt point each time a new packet passes by. It is generated for this packet only, which leads to an increased security, but also to a higher CPU usage. A new packet with the key as a payload and another new packet with the encrypted original payload are then created at the encrypt point. The encrypted data is generated by an XOR operation applied to the original payload with the generated key. The original packet is destroyed and explicit route information for the JVAR header option field is added to the two newly created IP packets. Therefore they will use the different predefined path routes through the simulated network. At the decrypt point, the first packet is being cached in a queue until its counterpart arrives. Packets from different key / encrypted payload packet pairs can be distinguished by the sequence number field in the JVAR header protocol. The original data is then restored with the same XOR operation of the key with the encrypted data. The restored original IP packet can then be sent through the rest of the network path until it arrives at the recipient. For another example of the "split crypt method" see also [Bro00]. See also figure 23 on page 55.

Figure 24 on page 55 shows how and where the packet with the original data gets split up into two new packets. They travel on two different pathways through the network until they get reassembled back to the original data packet. The restored original packet is then sent normally to the client, which receives it. As explained in section 3.8 on page 57 the whole process of generating the key, encrypting and decrypting the data without any graphical output displayed to the console and the observers takes about 188ms on the test system. This example uses only the out-of-band approach since the tunnel is setup on a certain stream before the actual network traffic occurs. The packets themselves do not provide any additional information for the active networking nodes.

3.7 MJPEG explicit multicast stream manipulation

Overview

A stream provider wants to send a (M)JPEG stream [BGB] to different clients, which have different network speed connections. The clients might not support the bandwidth the stream needs. This would lead to dropped frames for certain clients and those streams could get out of sync if only one multicast stream would

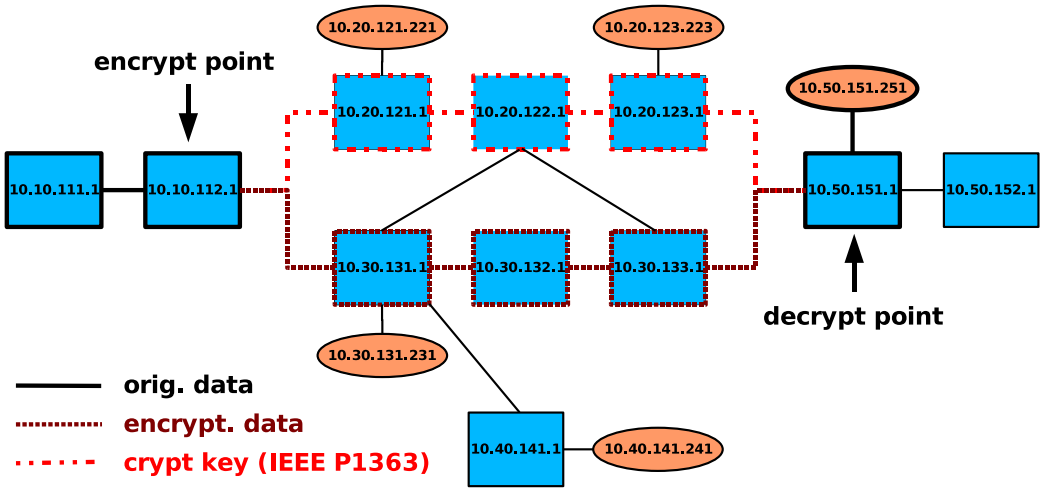

Figure 23: Split-crypt setup

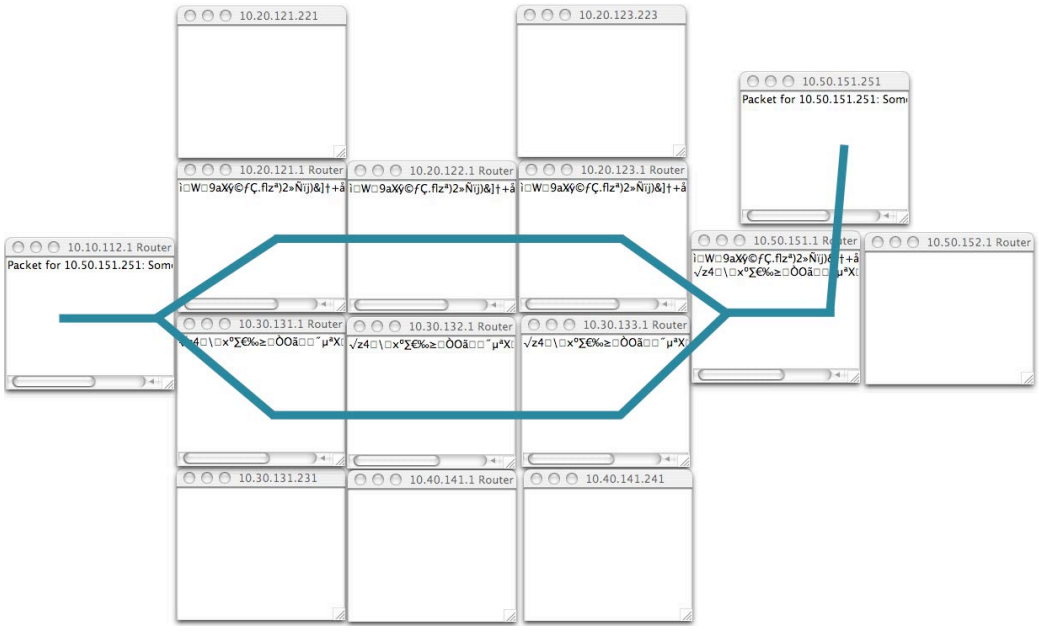

Figure 24: Screen-shot of split crypt example

be serving all the clients. With the JVAR framework, an explicit multicast [BFI+] stream with different flow modifications on critical node points could be introduced. An alternative without using JVAR would be for example multiple classic multicast streams for different connection speeds with different levels of qualities of the stream. This would lead to an overall higher bandwidth usage than the solution with the JVAR framework would offer.

Implementation

The network layout for this example is the same as explained in section 3.2 on page 43. A special JVAR protocol with the option field of the JVAR header holding the different IP destination addresses is used. The payload (images) gets modified by the FilterSets and sent as explicit multicast packets. The final packet for the recipient is again pure IP UDP (no special protocol needed). There are five modifying JVARCores with FilterSets and five clients that are receiving the image stream, which leads to 5x30 frames. In each modifying FilterSet, the JVAR header option field is checked for the IP addresses of the recipients. These addresses are then compared with the routing links of the routers. The packet is copied, if it has to be sent through different links of the router instead of only one link to reach all recipients defined in the JVAR header option field. All the destination addresses in the JVAR header option field are matched to the different links and added to the JVAR header option fields of the new IP packets. The allocation depends on the link through which the packet has to be sent through. At the end there are different explicit multicast packets that cover all recipients desired. The destination addresses of the new packets have been rewritten and the packets are then routed through the simulated network. This copying and matching to the links can happen at several JVAR nodes in the setup. In some of the FilterSets in the JVARCores the image contained in the IP payload gets modified before copying or routing it further. These modifications could include a conversion to black and white, a change of the image quality or a resizing of the image dimensions. Before the packets reach their final recipients, all JVAR related information (JVAR header option fields) gets removed and a normal UDP packet, which is addressed to the recipient is created. See also figure 25 on page 57.

Figure 26 on page 58 shows how the sender attached to the first router sends only one image through the second router (10.10.112.1). On its way through the network the image gets modified and duplicated in different ways. Changes of the resolution and the colour depth occur. The explicit multicast for the different routes and recipients continues until all clients have been served. As explained in section 3.8 on page 57 the whole 30 frame animation modification by the image processors on the benchmarking test system takes about 2781 ms which results

Figure 25: Explicit multicast of image with payload modification

in 54 frames processed per second. This example uses both the in-band and the out-of-band approach. First the nodes get setup with certain image modification algorithms (out-of-band) and then the packets (images) themselves (in-band) provide the multicast information.

3.8 Overview of performance measurements

The measurements are made on a computer with the following configuration: Athlon XP2800+ 166 FSB with 1 GB RAM (PC2700 CL 2.5), Operating System: Windows XP Professional, Mainboard with NForce2 Chipset. The log file output is disabled, no observers are attached to the JVARCores and the SUN J2SE 1.4.02 SDK is used. All measurements are made ten times, from which the average is taken. Especially fast or slow measurements differing significantly from the average are not used to build the final average. Possible system or Java garbage collector activity can provoke a significant slower execution of the example. Using this measurement falsifies the calculated average. The standard deviation for the different performance measurements presented did never exceed a maximum of five percent of the calculated average.

Simple copy: <1 ms

The duration for handling one single IP packet in one JVARCore is less than one ms. In the JVARCore 255 FilterSets are setup. They all have different IP mask rules and only the last one of these 255 matches. So the JVARCore has to calculate the match on 254 rules (of the FilterSets) before it finally finds the active

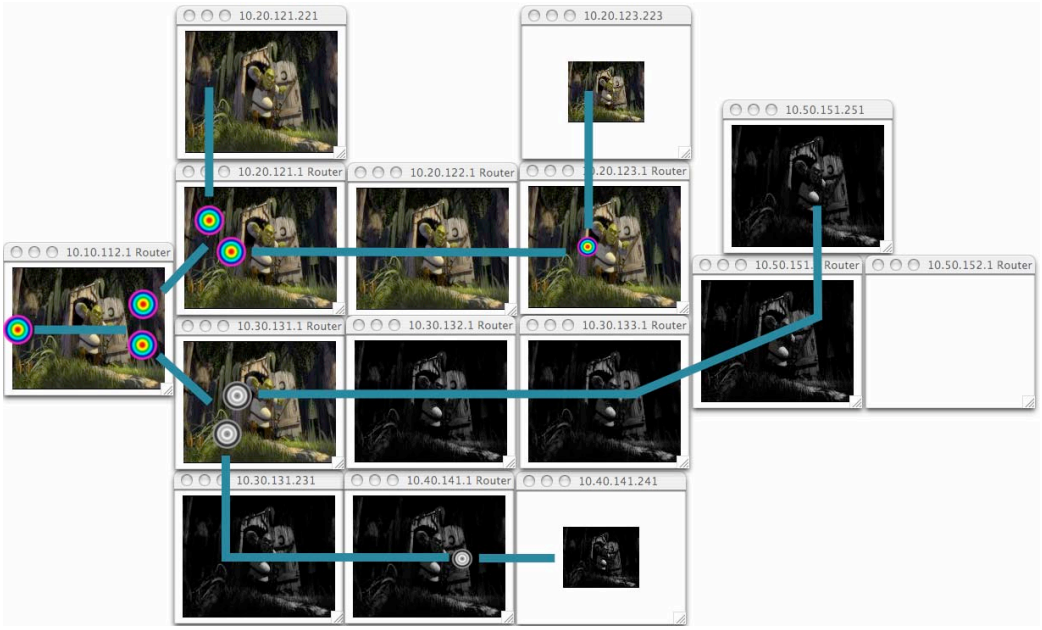

Figure 26: Screen-shot of MJPEG multicast example

matching FilterSet for the IP packet in question. The standard deviation for this measurement was 0.

Simple image manipulation: 61ms

A simple JPEG image with a size of 16.6 kilobyte is chosen. The conversion from a colour space (24bit) to black and white image is executed on the image by using the JPEGFilter in the JVAR framework. The final image after modification has the new size of 4.5 kilobyte. The original image resolution of 119x150 pixel is not modified. Only the image handling and manipulation is measured. No IP packet handling or JVARCore overhead is included in the measurement. The standard deviation for this measurement was 0.994.

Explicit route: 16 ms

An IP packet passes five JVARCores that analyse and modify the packet. Several routers without an active JVARCore are visited. Passing the packet from one router without a JVARCore attached to another is plain reference passing in Java. The overhead for this reference passing is negligible, the IP packet is not handed as a copy from a router to another, but only the memory reference is passed. Since the complete IP packet does not have to be copied and not sent through a media like a network, the performance is augmented drastically compared to a distributed

setup with different computers involved. Having only reference passing instead of copying objects helps only measuring the routing analysis and packet modification actions. The time for the packet handling between routers is not added to the measurement since the reference passing is not measurable and hence negligible. The JVAR header option field of the IP packet that is sent trough the network holds five IP addresses of routers that have to be visited. Each JVARCore at these five routers has to remove the first address in the JVAR header option field and uses it as the new destination address for the packet (rewrite of destination address). The standard deviation for this measurement was 0.568.

SplitCrypt tunnel: 188 ms

The setup consists besides the routers of one split JVARCore (encrypt point) and one reassemble JVARCore (decrypt point). The payload of the IP packet (in plain text) is "Packet for 10.50.151.251: Some text - And more". At the encrypt point, a random key with the same size like the payload is generated. Two new packets are created, where one is holding the generated key. The other holds the encrypted data, which is created by applying the XOR operation on the payload with the generated key. An explicit route JVAR header is added to the IP packets to send them through separate network paths. The two packets are sent through several routers in the network setup. The packet passing from one router to another is negligible for the measurement, since this is nothing else than memory reference passing. The first arriving packet is stored at the decrypt point, which is a negligible operation for the measurement. The original data gets restored when the second packet arrives. The original packet is rebuild by the XOR operation of the key with the encrypted data packet. For another example of the "split crypt method" see also [Bro00]. The standard deviation for this measurement was 9.141.

Mixed JPEGMulticast: 54 frames per second

The network setup has five modifying JVARCores and five clients, which receive a MJPEG image stream. 5x30 frames (=150 frames) for effective fps (frames per second) throughput calculation are sent through the setup. Three JVARCores (of the five) only duplicate the packets and recalculate the explicit multicast receiver addresses in the JVAR header option field and rewrite the destination addresses of those packets. Four JVARCores (of the five) modify the image with a black and white conversion or by resizing the image and might also do the duplication, the JVAR header option field recalculation and the address rewriting. The runtime of the complete measurement is 2781 ms which results in 11 frames processed per second per JVARCore and in 54 frames processed per second in the setup overall. The standard deviation for this measurement was 40.719.

4 Outlook

4.1 Filter / FilterSet extensions

In the following sections some possible extensions of the JVAR framework, that could be implemented based on the existing framework, are shown. Some other ideas of how to deal with security issues are also presented.

Using the native resources of the hardware and operating system of the machine on which a JVARCore is running would be another possibility to improve the performance. However, this would have the disadvantage of disabling the platform independence, which was achieved by using JAVA. Certain special OS functions or hardware components would be expected to execute such a platform specific active networking node or component. Some of these disadvantages could be avoided by presenting alternatives for the specialised functions. If in a JVARCore the hardware accelerations or some native modules and functions could not be accessed, an alternative platform independent solution would be used.

Also some alternatives for class loading are presented. Other kind of server communication as well as the earlier discussed security issues are also addressed.

Basic payload modification in native C or in hardware

Some methods to manipulate the payload could be written in a native language of the hardware / computer on which a JVARCore would run. If for example a fast compression algorithm is known in native C, it could be included by using JNI [Lia99] (Java Native Interface).

If a hardware implemented encryption board would be available in the system, it could also be used by JNI or directly by Java interfaces (if provided). All these actions would though decrease the portability, since the platform independence of Java could not be guaranteed anymore.

JIT and specialised compilation of Filters / FilterSets

JIT (just in time) compilation of a class could be done at the class server. Classes could be specially compiled for a specific JVARCore with fixed values for the IP addresses, keys and other parameters, which then would be directly put in the source code. Finally code generators based on templates for building a class could be introduced to the JVAR framework.

4.2 Security extensions

In the following sections the **class server** represents a server or service providing the class to be loaded by remote JVARCores. The **requesting JVARCore** is the JVARCore, that wants to load a class from a foreign class server as described above.

Sign with keys (public + private)

A public and private key pair [RSA77] for each JVARCore could be introduced. All Filters and FilterSets could be signed / encrypted with the receiving JVARCore's public key before it would be sent to them. This would prevent manipulation by the "man in the middle". A "man in the middle attack" occurs on the way between receiver and sender when some third party picks up the message and manipulates it. This way, all communication (between JVARCores, authentication servers, etc.) could be signed / encrypted by the public key of the recipient. Only the requesting JVARCore would have the private key to decrypt the packet. Therefore a packet sent through the network could not be modified on the way to the requesting JVARCore, since it cannot be decrypted by other parties.

There are different ways how the public / private key paradigm could be used with the class loading mechanism. Either the public key would be sent on each request, where the JVARCore requesting a class at the class providing server would send its public key together with the request. Or the public key would already be known by the class server, because the JVARCore upon initialising itself or on first contact with a class loading request would send its private key to the class server, which then would cache the key. If the key transmission would happen on initialisation, the JVARCore would have to know the class server's address during its startup.

The key pair could either be hard coded in the JVARCore or it could be generated at startup of the JVARCore in order to increase security. Validity of the key pair could be for the whole lifetime of the JVARCore instance or a certain lifetime could be assigned to a key pair. For example every 30 minutes a new key pair would be generated, but then the newly generated keys would have to be sent to the class server upon the next class loading request.

Authentication servers

All requests could be checked first by a global authentication server before they would be declared valid. When a requesting JVARCore would receive a remote class loading request, it would check if the class server could be trusted by asking

the global authentication server. Class servers would have to do handshakes with the global authentication server after they initialised themselves and also periodically during their runtime. The global authentication server could hold the public keys of the class servers, so it could send those to the JVARCores requesting the information. This would lead to a secure request of the JVARCore to the class server, since it would not be possible to modify the request on its way to the class server.

The communication between the authentication server and the requesting JVARCore could also be secured for example by public / private key pairs. The classes requested and then sent by the class servers remain encrypted and the check if a class server could be trusted would be done as well. However an additional advantage would be, that the requests sent to the authentication server would also be encrypted.

HTTPS / SSL

The class servers could be using HTTPS servers, which would encrypt their communication via SSL (Secure Socket Layer). A requesting JVARCore would have to request the class via HTTPS. The certification process used by SSL (used by HTTPS) would help to determine the trust level of a class server. No global authentication server would be needed, and the communication channel would be secure.

SSL encryption could also be used without HTTPS for example using other designated protocols. In this case the communication channel would have to be secured manually by SSL.

Compare checksums before class-loading

Instead of a JVARCore having to load the class on each class loading request, the JVARCore could first compare the checksum of the class. The class could already be locally available in the JVARCore. Either the class loading request would contain the checksum of the class to be loaded. Or the class loading JVARCore would first make a request for the checksum at the class loading server, before it would decide to load the class completely.

5 Summary

In chapter 1 classical networks and their behaviour have been presented. Their disadvantages have been explained as well as their limitations that arise from their statical nature. A short introduction to active networking has been given, by showing the different kind of approaches to it. The in-band and out-of-band approaches were illustrated as well as the limitations and disadvantages for each of them have been pointed out. Resource management issues were outlined and different examples with possible solution approaches have been shortly discussed. Issues concerning emulated virtual routers and networks were brought up, analysed and their usage and possibilities demonstrated. An overview of possible applications has been given included tunnelling, secure pathways, video-splitting, explicit multicast applications, explicit routing for reliability and performance issues, image and voice flow modifications and conversions, compression methods, usage based rerouting, agent handling and other interesting concepts like payload translation, correction or even payload filtering for parental control and classical TOS and QOS issues were discussed. Last but not least, an overview of other existing active networking applications written in Java was given followed by a presentation of the structure of this document.

Chapter 2 explained the approach used with the JVAR framework. It has been outlined, why a mixed approach of in-band and out-of-band has been practised and what both approaches would have had as disadvantages when only used on its own. Also the importance of using the wide spread standard IP /UDP protocol instead of a proprietary implemented protocol has been described. Java with its advantages to use it for this framework has been observed as well, taking also possible disadvantages in account. The JVAR framework has been explained in more detail. First the advantages of the framework were presented. The issue of being thread free has been described, the importance of a good documentation has been pointed out, the use of a logging framework was mentioned, the importance of having very efficient low level packet handling classes and to use a unit test framework has concluded the short overview. The basic setup of a JVAR node then has been presented. All those components were briefly described with their main duties and interconnections between them as well as the packet flow through such a setup. All main classes of the JVAR framework then have been analysed and explained in more detail with eventually some short code examples to help to understand the classes and their usage and behaviour being added to the description. To conclude the chapter, the arising security issues in an active network environment have been brought up and the difference between a safety and a security framework has been layout followed by a closer look at the implemented security and the logging framework.

Example active networking applications have been presented in chapter 3. After an overview, which has been including motivation issues and different arising or before hand occurring problems has been given, the implementation of each example scenario has been described in detail with additional images explaining the setups or actions. Basically two main setups, a simple network layout and a complex network layout, have been used to simulate the example scenarios. All these examples are provided with the JVAR framework and can be tested out, played with and adapted to other scenarios or network setups. Different simple and more complex applications have been used for the performance evaluation as well as some basic image manipulation and packet active component matching has been measured. All those measurements have been made on one single machine. Virtual networks have been built up for the analysis and measuring the speed. In order to simulate a complex active network application the achieved performance seemed to be reasonable.

The outlook in chapter 4 presented some extension possibilities of the JVAR framework. Additional usage of native or in other languages provided resources, algorithms or accelerations were observed. JIT and special compilation were discussed to provide more platform specific advantages and code optimisations. Security extensions by using the public / private key paradigm has been taken into closer consideration. Also the possible introduction of authentication servers has been examined. The increase of security by using HTTPS and SSL was added to the discussion. To finish the chapter, the possibility of comparing checksums before loading classes in order to increase the performance was analysed.

As a conclusion, the JVAR framework can be considered as very open, easy to understand, very extensible and usable in real life as well as in simulated environments, hence working on pure IP and UDP packets without the need of a proprietary protocol. Different paradigms (in-band and out-of-band) have been put together to make a very reliable and flexible framework to work with. Also a focus has been on good documentation and platform independence as well as having the possibility to integrate or make it collaborate with different kind of real routers. A lot of programmed example applications and example classes have been provided to help building up other applications or extensions very easily and to understand the basic concepts of the interactions in a JVAR node.

A Class overview

A.1 Package listing of the JVAR framework

com.brogle.jvar.clients	simulated clients for demonstration
com.brogle.jvar.cores	core classes for JVAR and scheduling
com.brogle.jvar.demos	all demos provided with the JVAR framework
com.brogle.jvar.filters	filters for manipulating byte arrays
com.brogle.jvar.filters.jpeg	image related filters
com.brogle.jvar.filtersets	high level packet manipulation classes
com.brogle.jvar.handlers	handlers for communication interfaces
com.brogle.jvar.loaders	class loaders for remote class loading
com.brogle.jvar.observers	observers to view what is happening in a network
com.brogle.jvar.packets	all packet related classes (IP, JVAR)
com.brogle.jvar.routers	simulated routers for demonstration environment
com.brogle.jvar.servers	class servers providing classes for loaders
com.brogle.jvar.utils	helper classes and utilities used with JVAR

Now follows for each package a list of the included classes, with a short description of their duties.

A.2 com.brogle.jvar.clients

SimpleClient

Simulates a real client in a network and is connected to a router. Is needed for the programmed examples of active networking applications to show the behaviour of the framework. Can have observers attached to it to view what is happening inside a client.

A.3 com.brogle.jvar.cores

JVARCore

This is the main core class that handles all manipulation, task handling, IP matching, class loading and unloading. Has methods, which analyse packets and match them to a corresponding FilterSet or add the tasks being sent to the TaskCore.

TaskCore

A TaskCore holds all the tasks that have to be executed. Calls are being made by the JVARCore to get next task that has to be executed. The JVARCore adds the tasks it receives from other JVARCores to its TaskCore.

A.4 com.brogle.jvar.demos

AbstractDemo

A helper class for all programmed examples of active networking that implements the basic example handling and provides the round robin execution / triggering of JVARCores. All other examples have to inherit this class. The AbstractDemo checks for packets to be handled by a JVARCore and also triggers all other actions outside the JVARCore.

BasicDemoSetup

This is the base class for all programmed simple examples of active networking. Sets up the routers, RouterHandlers, clients, observes and JVARCores for the simple example network layout. All other programmed examples of active networking based on this network layout inherit this class.

ComplexDemo

This is a basic demonstration of the complex example network setup. Sends an IP packet with text payload to all clients in the simulated network. Observers are attached to each client, to follow the way of the IP packet through the simulated network layout.

ComplexDemoSetup

This is the base class for all programmed complex examples of active networking. Sets up the routers, RouterHandlers, clients, observers and JVARCores for the complex network demo layout. All other demos based on this network layout will inherit this class.

ExplicitRouteDemo

This is a demonstration of explicit routing with the JVAR framework. It sends a text payload through the simulated network setup. On the observers attached to the demo, the path of the packet can be followed and it will show, that it does not follow the default route, but the path defined in the JVAR header option field.

GZIPDemo

This example of active networking shows how a text payload gets compressed and decompressed on certain nodes in the simulated network setup. Builds a GZIP tunnel through the network. Observers either show the readable text or the compressed payload.

JPEGDemo

This is a demonstration of the image manipulation with the JVAR framework. Does black and white conversion and image resize using the JVAR framework. Observers will show the different kind of manipulated images on the corresponding JVARCores they are attached to.

MulticastDemo

This is an example of explicit multicasting using the JVAR protocol with the JVAR header option field. Observers will show that only one packet is injected into the simulated network layout, but that on the corresponding nodes it will get copied, multiplied and routed further.

MulticastJPEGDemo

This example application shows the combination of explicit multicast and image manipulation using the JVAR framework. Observers will show how packets will get copied and multiplied, as well as the manipulations of the images that have taken place on the corresponding JVARCores they are attached to.

MulticastMJPEGDemo

In this demonstration, a 30 frame video sequence is sent through and modified in the simulated network layout. Observers will show, how different clients will receive the video in different qualities. All starts from one single high quality feed. The points in the simulated network, which do the the multiplication / copying of the packets and image modifications are also displayed.

SimpleDemo

This is a basic demonstration of the simple example network setup. Shows the use of the base classes for an example and how these demonstration classes have to be implemented. Also shows how to use observers, SimpleFilters / SimpleFilterSets and the use of remote class loading and remote FilterSet / Filter setup. The output

is written to the console and a log file. The observers show the payload of the packets, which the attached clients or JVARCores receive.

SplitCryptDemo

A more complex example of active networking, which uses explicit route and packet splitting to encrypt the payload with a key. The encrypted packet and the key are sent over separate paths through the simulated network. At a certain node in the simulated network layout, the two packets get merged again to the original data. Observers show the content of the different packets (either original data, encrypted data or the generated key).

A.5 com.brogle.jvar.filters

AbstractFilter

This is the base class for all filters. It holds basic filter handling functions and provides filter chaining. All filter implementations have to inherit from this class to match a common interface and to allow the integration of unknown filters at runtime. This integration is done with the help of polymorphism[11].

DuplicateFilter

A simple duplicating filter used as an example how to create real filters. Duplicates the byte array it receives. Since filters can return more than only the one byte array they received, this filter also shows how this has to be done and handled with.

DuplicateFilterTest

The test class for DuplicateFilter that shows how test classes should be written for real filters. Checks if from a single byte array input two equal byte arrays (equal in contents, not references) will be produced as an output.

GZIPFilterTest

This is the test class for GZIPInputFilter and GZIPOutputFilter. Checks if a simple byte array containing some text will be compressed and successfully decompressed having the same content as the input.

[11] In object-oriented programming, polymorphism (from the Greek meaning "having multiple forms") is the characteristic of being able to assign a different meaning or usage to something in different contexts - specifically, to allow an entity such as a variable, a function, or an object to have more than one form. There are several different kinds of polymorphism.

GZIPInputFilter

Decompresses the byte array it receives with the free GZIP compression algorithm. Used by the GZIPInputFilterSet.

GZIPOutputFilter

Compresses the byte array it receives with the free GZIP compression algorithm. Used by the GZIPOutputFilterSet.

ObjectInputFilter

Receives a serialised object in a byte array and deserialises it back to the object. Could be used for sending or migrating objects (mainly Filters and FilterSets) with a certain state from one JVARCore to another (remote) JVARCore.

ObjectOutputFilter

Receives an object and serialises it to a byte array. Could be used for receiving objects (mainly Filters and FilterSets) with a certain state from other (remote) JVARCores.

SimpleFilter

An example filter implementation that only passes the data it receives to the next filter (or receiver). Helps to understand how to implement real filters. Used for the SimpleDemo.

SimpleFilterTest

The test class for SimpleFilter that helps to understand how to implement real filter tests. Tests only if the byte array sent as input contains the same as the output provided by the SimpleFilter.

StripSizeHeaderFilter

A filter that removes a certain header length from a byte array. Mainly used as an example of how to remove obsolete parts from payload data.

StripSizeHeaderFilterTest

The test class for StripSizeHeaderFilter that checks if a certain amount of a byte array is removed from the input.

A.6 com.brogle.jvar.filters.jpeg

JPEGFilterAbstract

This is the base class for all image manipulation filters. All filters doing image manipulation have to inherit from this class, which provides basic image serialisation and deserialisation from and to a byte array, which holds a valid JPEG image. Also provides basic functionality for all filters to change the image quality (compression factor), which all other filters automatically will also provided additionally to their specialised manipulation functionalities.

JPEGFilterBW

A colour to black and white conversion JPEG filter that changes the colour depth of a true colour (24 bit palette) image to a (8 bit) black and white image.

JPEGFilterBWTest

The test class for JPEGFilterBW that checks if the output image is smaller in size than the input image.

JPEGFilterResize

An image resize JPEG filter that resizes the image by a certain factor (0.0 - 1.0) on both dimensions equally.

JPEGFilterResizeTest

The test class for JPEGFilterResize that checks if the output image is smaller in size than the input image.

JPEGFilterSimple

A simple image filter to show how to implement real image manipulation filters. This filter keeps the image dimensions and colour depth, but only changes the compression (quality) factor of it. Shows also how to use the basic manipulation methods inherited from the JPEGFilterAbstract.

JPEGFilterTest

An example of a test class for image manipulation filters that shows how to implement JPEG filter test classes. Checks if the output image is smaller in size than the input image.

A.7 com.brogle.jvar.filtersets

AbstractFilterSet

The base class for all FilterSets that provides a communication interface to the TaskCore and router to which the JVARCore holding the FilterSet is connected to. The AbstractFilterSet also gives access to the TaskCore via the JVARCore, so that FilterSets can also create new tasks.

GZIPInputFilterSet

A FilterSet for IP payload GZIP decompression. Used to setup GZIP tunnels. Uses the GZIPInputFilter.

GZIPOutputFilterSet

A FilterSet for IP payload GZIP compression that is used to setup GZIP tunnels. Uses the GZIPOutputFilter.

JPEGBWFilterSet

This is a colour to black and white conversion JPEG FilterSet. Uses the JPEG-FilterBW. Extracts the JPEG image information from the IP packet and hands it to the Filter for image manipulation and creates the new IP packet with the new manipulated image.

JPEGResizeFilterSet

An image resize JPEG FilterSet that uses the JPEGFilterResize. Extracts the JPEG image information from the IP packet and hands it to the Filter for image manipulation and creates the new IP packet with the new manipulated image.

JVARExplicitRouteFilterSet

An explicit route FilterSet that compares local router IP address with the IP packet destination address. If they match, it rewrites the IP header destination address and modifies the JVAR header option field by removing the local address from the list of IP addresses that have to be visited.

JVARJPEGResizeMultiCastFilterSet

This is a combination of explicit multicast and the JPEG resize FilterSet. The explicit multicast part copies packets and rewrites IP destination addresses and JVAR

header option fields depending on local IP address of the router. The JPEG resize FilterSet part manipulates the newly created image payload of the IP packets. See also JVARExpliciteRouteFilterSet and JPEGResizeFilterSet.

JVARMultiCastFilterSet

Multiplies IP packets and rewrites IP header destination address field as well as the JVAR header option field depending on the local IP address of the router. Before reaching the client, packets will be cleaned of any JVAR specific protocol additions, so that the recipient receives an ordinary UDP or IP packet.

SplitDecryptFilterSet

Reconstructs the original IP packet from two IP packets. One of the two IP packets, which was created from from the original packet, contains a key. This key was used to create the second packet, which contains the encrypted original data. The two packets were sent on different paths through the network. An XOR function is applied on the encrypted data with the key to reconstruct the original data.

SplitEncryptFilterSet

Splits and encrypts an IP packet into two new IP packets and transmits them via explicit routing with additional JVAR header information by two separate paths through the network. The first packet holds the key that has been used for the encryption. The other packet holds the encrypted original payload, that has been created by using the XOR function on the original data with the generated key.

SimpleFilterSet

An example of an implementation of a FilterSet that shows how to create FilterSets. Uses the SimpleFilter and passes only the payload through the Filter. The payload and the packet will not be modified during this process.

SimpleFilterSetTest

The test class for SimpleFilterSet that shows how to write FilterSet test classes. Checks if the input packet will correspond in content to the output packet of the SimpleFilterSet.

A.8 com.brogle.jvar.handlers

AbstractRouterHandler

The base class for all RouterHandlers that provides a basic communication interface between the JVARCores and the routers they are connected to. All router handler implementations have to inherit from this class to provide a basic interface that the JVARCores then can access. Extensive test classes should be written for all router handler implementations to guarantee the proper functionality and reliability of the communication interface for a certain router.

RouterHandler

The router handler for the simulated router written in Java, which comes with the JVAR framework. Shows also, how to implement a RouterHandler for other routers. Fragmentation and defragmentation of packets have to be done in the RouterHandler layer. Since the simulated router also works with the JVAR IP packet implementation, the communication between the router and the JVAR-Framework is mainly reduced to reference passing. This makes the overhead for passing packets through the RouterHandler negligible.

RouterHandlerTest

The Test class for the RouterHandler that checks if packets sent through the router handler will reach the router and if receiving from the router via the router handler works. This simple test class also shows how to implement any other router handler test class.

A.9 com.brogle.jvar.loaders

AbstractLoader

This is the base class for all class loading implementations. It provides the class loading interface for a JVARCore. All other class loader implementations have to inherit from this class to provide the same interface. This also allows to exchange the class loader at runtime with other not yet known or remotely loaded class loaders.

SocketClassLoader

An implementation of a class loader that loads classes through sockets by connecting to a SocketClassServer. The class loading is triggered by the JVARCore. If the JVARCore receives a class loading task, it passes the class loading request

to the SocketClassLoader for execution. The port of the socket is defined at start up of the JVARCore. See also the SocketClassServer.

SocketClassLoaderTest

The test class for SocketClassLoader that checks if the right class gets loaded after a class loading request and if a successful connection to a SocketClassServer can be made.

A.10 com.brogle.jvar.observers

AbstractObserver

This is the base class for all observers. It provides GUI and console output handling. All observer implementations have to inherit from this class. Observers then can be connected to a JVARCore or a Client via this common inherited interface.

ImageObserver

An observer for image payload that shows the image in the GUI or the image information at the console. Strips JVAR header information before displaying the information. Used by the demos that deal with images or whenever image manipulation wants to be followed. Shows GIF or JPEG images out of a byte array.

ImageObserverTest

This test class for an ImageObserver checks, if an image provided in a byte array can be displayed successfully.

PayloadObserver

The observer for text payload that shows the text payload in GUI or console. Strips JVAR header information before displaying the information. Used by the programmed examples dealing with text payload or whenever text payload manipulation wants to be followed. Shows the text in the payload on the console and in the GUI window.

A.11 com.brogle.jvar.packets

IP

This is an implementation of the IP protocol with UDP support and helper functions for IP header manipulation. Is used throughout the whole JVAR framework, since it represents the IP packets the JVAR frameworks receives from the routers. This class is also used by the router implementation written in Java for simulating complex network layouts on one computer.

IPTest

A test class for IP class that tests all header setting methods as well as all data retrieving methods. Also UDP manipulation is tested.

JVAR

This class provides the JVAR header protocol information with helper functions. Implements all JVAR related protocol extensions in the JVAR header and its option field. Helper functions include retrieving and adding IP addresses from the option field.

JVARTest

The test class for JVAR class that tests all protocol extensions and specially the helper functions to work as expected.

Packet

This is base class for all protocols and packets. It provides helper functions for getting and setting bytes, words, etc. from and to a byte array. All packet related classes (IP and JVAR) inherit this class to have the benefit of the manipulation methods for the byte array handling.

A.12 com.brogle.jvar.routers

Router

A simulated router to show the use of the JVAR framework and to simulate real networks on one computer. Is mainly used by the demos provided with the JVAR framework. Uses the RouterHandler class so that JVARCores can actually communicate with this router implementation. The router itself is very efficient and

fast, since passing IP addresses from routers to other routers or from and to JVARCores is done by passing references.

RouterTest

The test class for the Router class that tests the routing functionality, and the router interconnectivity.

A.13 com.brogle.jvar.servers

SocketClassServer

A class server that provides classes on demand for JVARCores wanting to load them. Serves the SocketClassLoader. A SocketClassServer is independent of a JVARCore running on a computer. Therefore it can be run on a separate machine (or on the same) as a JVAR independent service.

SocketClassServerTest

This is the test class for the SocketClassServer. Tests if the server provides the right classes for a loading request by a SocketClassLoader and if it can be setup successfully on a machine.

A.14 com.brogle.jvar.utils

ComplexIPMask

A helper class for matching different IP header parts with different wild-card possibilities. Mainly used by JVARCores to find matching FilterSets for an IP packet. Matching is done by comparing addresses, TOS fields, UDP ports either strictly, by using masks on addresses or by wild-cards (anything matches).

ComplexIPMaskTest

The test class for ComplexIPMask that checks all implemented matching mechanisms including the wild-card option. Also all implemented methods to retrieve existing IP header data out of IP packets and the constructor for the the ComplexIPMasks are tested.

Constants

Holds all static constants for the JVAR framework, for example maximum IP packet length. Can be accessed from any class within the JVAR framework without actually instantiating it as an object.

IPAdr

A helper class to extract IP address information from IP packet objects and from strings holding IP addresses. Also matches IP addresses with network masks. Mainly used by FilterSets and by the routers written in Java to find the next matching router for an IP packet.

IPAdrTest

The test class for the IPAdr class that tests the different matching methods and constructors to retrieve IP addresses out of IP classes and strings.

IPFactory

A helper class to generate IP packets (instances of the class IP). Provides methods to create easily IP classes by handing string IP addresses and options or partial IP information. This will lead to a valid IP class, which holds all necessary information to be be used within the JVAR framework. Used by the demos and the test classes to create test and demo IP packets.

JPEGFilterApplier

A helper class to apply image manipulation filters to IP packets containing image JPEG payload. Mainly used for all image manipulation related test classes to reduce duplicated code and guarantee same conditions for all tests.

Task

Tasks that represent different actions to be executed by a JVARCore. Tasks are stored in the TaskCore and are executed by the JVARCore. Tasks can be serialised and sent to other (remote) JVARCores where they will be deserialised, stored and executed as needed.

TaskTest

The test class for Task class that checks, if tasks can be successfully serialised and then deserialised as well as instantiates the different kind of Tasks.

TestHelper

A helper class for all test classes that provides example IP packets to run the tests with. Helps to have same test conditions for the different tests and getting the test IP packets very easily.

B Source code examples

In this chapter some parts of the source code are shown, which help to get a better understanding of the framework.

B.1 JVARCore

Below the constructor of a JVARCore is shown, giving an idea, what is needed to setup a running JVARCore successfully:

```
/**
 * Constructor connects to JVARCore to a router handler
 * @param routerHandler Router handler to which this
 *                      JVARCore is connected
 */
public JVARCore(AbstractRouterHandler routerHandler,
                String coreName) {
    // initialise all hashtables
    filtersets = new Hashtable();
    observers = new Hashtable();
    // connect to router handler
    routerhandler = routerHandler;
    taskcore = new TaskCore();
    // init task mask
    // (source does not matter, but destination
    // and protocol have to match)
    taskCIPM = new ComplexIPMask(
                new IPAdr("0.0.0.0/0.0.0.0"),
                routerHandler.getLocalIP(),222, -1);
    // set coreName
    this.coreName = coreName+": ";
}
```

The main matching and selection is done in one method, the "Matchmaker":

```
/**
 * Tries to get the next packet from the router handler
 * and processes it
 * @return true if a packet was processed, else false
 */
public boolean processNext() {
    IP ip = routerhandler.getNextIPPacket();
```

```
if (ip!=null) {
    // get complex ip mask from ip packet
    ComplexIPMask cipm = new ComplexIPMask(ip);
    // find observer
    if (processByObserver(ip, cipm))
        logger.info("found an observer");
    else logger.info(coreName
                     + "no observer found");
    // check if it is a task
    // (this could also be done
    // by a filterset if wanted)
    if (taskCIPM.matches(cipm)) {
            // add task to TaskCore
            taskcore.addTask(ip.payload);
            logger.info(coreName
                        + "Found a matching task");
            // stop further processing
            // of task ip packet
            return true;
    }
    // find filter
    IP[] processedIPs =
      processByFilterset(ip, cipm);
    // check if filter has been applied
    if (processedIPs!=null) {
            logger.info(coreName + "found a filterset");
    }
    // or else copy the original packet
    // to the processed array
    else {
            logger.info(coreName +
              "no filterset found");
            processedIPs = new IP[1];
            processedIPs[0] = ip;
    }
    // route packets
    for (int i=0; i<processedIPs.length; i++) {
      routerhandler.routeIPPacket(processedIPs[i]);
    }
    // got a packet from router handler,
    // so return true
```

```
        return true;
}
```

B.2 Task

The following types of tasks exist (taken from the source code directly):

```
/**
 * Task type for adding filtersets
 */
public static int TASK_FILTERSET_ADD = 0;

/**
 * Task type for removing filtersets
 */
public static int TASK_FILTERSET_REMOVE = 1;

/**
 * Task type to notify a class
 */
public static int TASK_NOTIFY = 2;

/**
 * Task type to load a class from a remote server
 */
public static int TASK_LOADCLASS = 3;

/**
 * Task to send an IP packet
 */
public static int TASK_SENDIPPACKET = 4;

/**
 * Task to do nothing at all
 */
public static int TASK_DUMMY = 5;
```

Tasks have various parameters, which hold objects with different kind of meanings depending on the task type:

```
/**
 * Constructor to create a task
```

```
 * @param executeInMilliSecs Time when task has to be
 *                           executed from now
 * @param repeat Times the task has to be repeated
 *               (-1 = endless)
 * @param tasktype Type of the task
 * @param taskObject The task object depending
 *                   on the type
 * Parameters for task objects depending on
 * the task type:
 * TASK_NOTIFY:
 *   taskObject1 = object: reference to object being
 *                         notified
 *   taskObject2 = nothing
 * TASK_DUMMY:
 *   taskObject1 = nothing
 *   taskObject2 = nothing
 * TASK_FILTERSET_ADD:
 *   taskObject1 = string: name of filterset class
 *   taskObject2 = ComplexIPMask: complex ip mask for
 *                                the filterset
 * TASK_FILTERSET_REMOVE:
 *   taskObject1 = ComplexIPMask: mask of the filterset
 *                                to remove
 *   taskObject2 = nothing
 * TASK_LOADCLASS:
 *   taskObject1 = String: Name / Address of class
 *                         providing server
 *   taskObject2 = String: Name of the class
 *                         to be loaded
 * TASK_SENDIPPACKET:
 *   taskObject1 = IP: IP packet that has to be sent
 *   taskObject2 = nothing
 */
public Task(long executeInMilliSecs, int repeat,
            int tasktype, Object taskObject1,
            Object taskObject2) {
        this.starttime =
          Calendar.getInstance().getTimeInMillis();
        this.executeInMilliSecs = executeInMilliSecs;
        this.repeat = repeat;
        this.tasktype = tasktype;
```

```
        this.taskObject1 = taskObject1;
        this.taskObject2 = taskObject2;
}
```

Example of a simple task, that loads a filter from a ClassServer:

```
Task taskLoadFilter =
  new Task(0, 1, Task.TASK_LOADCLASS,
           "localhost:8888,
           "com.brogle.jvar.filters.SimpleFilter");

// add the serialised task to an IP payload
IP ipTaskLoadFilter =
  IPFactory.getIP(192,168,0,1,192,168,10,1,222,0,
                  taskLoadFilter.getSerializedObject());

// route the IP packet via a RouterHandler (rh0)
rh0.routeIPPacket(ipTaskLoadFilter);
```

B.3 AbstractClassLoader

The real implementation of a class loader has to inherit from the AbstractClass-Loader and has to provide a custom constructor and a class loading method:

```
/**
 * Constructor gets parameter to which server
 * it should connect
 * @param ServerSource Source indicating the server
 */
public AbstractLoader(String serverSource) {
this.serverSource = serverSource;
}

/**
 * Fetches a class from a server specified by
 * constructor and saves it locally for further use
 * @param className Fully qualified class name
 *                  to be fetched
 */
public abstract boolean getClass(String className);
}
```

B.4 SocketClassServer

The implementation of the SocketClassServer or any other class server does not have to inherit from any abstract super class, since any kind of service can be provided. Only the corresponding class loader (the class loading class) has to be adapted to the class server (and to the communication interface it provides) it has to connect to.

The example SocketClassServer provided offers classes to load via a socket on a specific port:

```
/**
 * Constructor initialises the socket server
 * @param port Port on which the server should listen
 */
public SocketClassServer (int port) {
    try {
        // create new socket server bound to port
        serversocket = new ServerSocket(port);
        logger.info("Created server socket on port: "
                    + port);
    }
    catch (Exception e) {
        logger.warn("Could not create socket server "
                    + "on port:" + port);
    }
}
```

B.5 RouterHandler

To show how easy the implementation of a router new access is, the basic methods of the abstract super class, which have to be overwritten are listed below:

```
/**
 * Returns the local IP address of the router which
 * the handler is connected to
 * @return ipAdr Local IP address of connected router
 */
public abstract IPAdr getLocalIP();

/**
 * Returns all routing table entries of the router
```

```
 * to which a router handler is connected to,
 * could be realised in the implementing class
 * by the help of sendRouterCommand
 * @return Routing table of the router connected to
 */
public abstract Vector getRoutes();

/**
 * Returns next IP packet to handle, defragmentation
 * should be made here if needed
 * @return IP Next IP packet to handle
 *            from connected router
 */
public abstract IP getNextIPPacket();

/**
 * Receives next packet for the router to route,
 * fragmentation should be made here if needed
 */
public abstract void routeIPPacket(IP aPacket);

/**
 * Sends a command or information to the router
 * @param aCommand byte array holding the command
 *                 or information
 * @return Answer of the router to this command
 */
public abstract byte[] sendRouterCommand(
                         byte[] aCommand);

/**
 * Sends a filter configuration to the router, that
 * handler would like to receive all packets
 * matching the filter setup sent
 * @param complexIPMask Filter setup for router
 *                      to be configured
 * @see #removeFilter(ComplexIPMask)
 */
public abstract void setupFilter(ComplexIPMask
                       complexIPMask);
```

```
/**
 * Removes a filter configuration from the router
 * @param complexIPMask Filter setup for router
 *                      to be configured
 * @see #setupFilter(ComplexIPMask)
 */
public abstract void removeFilter(ComplexIPMask
                      complexIPMask);
```

B.6 ComplexIPMask

There are two main constructors provided.

Either a custom ComplexIPMask is created by setting all the mentioned IP matching parts manually:

```
public ComplexIPMask(IPAdr source, IPAdr dest,
                     int protocol, int tos,
                     int udpsrc, int udpdst)
```

Or all necessary information is extracted directly from an existing IP packet:

```
public ComplexIPMask(IP ip)
```

The check itself is done very easily and it aborts on the first mismatch all further matching tries (better performance):

```
/**
 * Compares 2 ComplexIPMasks and checks if they match
 * (with wild-cards)
 */
public boolean matches(ComplexIPMask other) {
    // check for matching tos
    if (this.tos!=-1 && this.tos!=other.tos)
      return false;
    // check for matching protocol
    if (this.protocol!=-1 && this.protocol!=
        other.protocol) return false;
    // check for matching destination
    if (!this.dest.isMatching(other.dest))
      return false;
    // check for matching source
    if (!this.source.isMatching(other.source))
```

```
      return false;
    // check for udp parts only if protocol = 17
    if (this.protocol == 17) {
        // check udp src
        if (this.udpsrc!=-1 && this.udpsrc !=
            other.udpsrc)
          return false;
        // check udp dst
        if (this.udpdst!=-1 && this.udpdst !=
            other.udpdst)
          return false;
    }
    // all match
    return true;
}
```

C Log4J

To write to a log file or to the std out[12], the LOG4J framework [Gul02] from the Apache Software Foundation has been used.

It can be easily configured (and also turned off) and is known to be very very resource effective and time saving.

Only one line has to be inserted to make a class logging aware:

```
protected static Logger logger =
   Logger.getLogger(ClassName.class);
```

Almost all classes in the JVAR framework do some kind of logging (console and / or log-file output). Since each of the classes belong to a certain package, and all of them have distinguished logging ids (the class name itself), logging can be configured on different levels. Exceptions can directly be handed over to the logging framework to format them nicely as logged output. The whole log file / console output can be formated in many variations (with fully classified classname, date, time, level, etc).

In the file `logoptions.properties`, logging can be set to different levels (debug, warn, info, fatal, error, log), in order to show only logging messages of that kind and above the configured level. This way, for example only fatal messages would be logged. On the other hand, logging can be configured on any class or per package. See the example configuration (`logoptions.properties`), that is used for the JVAR framework by default, which includes logging to the std out and to a log-file with a rollover. For the rollover settings, the maximum size of a log-file and the maximum number of backup-files are defined. After the maximum file size has been reached and all numbers of backups allowed are used up, the oldest backup log file gets deleted and a new one for current logging is then created.

The example configuration (stored in the file called logoptions.properties):

```
# log to console (std out), debug (std err) and
# file appender R
log4j.rootLogger=debug, stdout, R

# configure standard out
```

[12] Standard output to console or error stream provided by the operating system

```
log4j.appender.stdout=org.apache.log4j.ConsoleAppender
log4j.appender.stdout.layout=
    org.apache.log4j.PatternLayout
log4j.appender.stdout.layout.ConversionPattern=
   %d [%t] %-5p %c - %m%n

# configure customised file appender R
log4j.appender.R=org.apache.log4j.RollingFileAppender
log4j.appender.R.File=jvar.log
log4j.appender.R.MaxFileSize=100KB
log4j.appender.R.MaxBackupIndex=1
log4j.appender.R.layout=org.apache.log4j.PatternLayout
log4j.appender.R.layout.ConversionPattern=
   %d [%t] %-5p %c - %m%n

# Print only messages of level WARN or above
# for this package
log4j.logger.com.brogle.jvar.filters.jpeg=WARN
```

For more logging options and examples of log file configurations, see [Gul03].

References

[AB+] D. Scott Alexander, Bob Braden, et al. Active network encapsulation protocol (ANEP). Request for Comments: DRAFT, Category: Experimental. Document can be found at: http://www.cis.upenn.edu/~switchware/ANEP/docs/ANEP.txt, visited on February 17th 2004.

[Ane01] George Anescu. A C++ implementation of the blowfish encryption/decryption method, September 2001. Webpage can be found at: http://codeproject.com/cpp/blowfish.asp, visited on February 17th 2004.

[Bal] Y. Mario Baldi. End-to-end delay of videoconferencing over packet switched networks. IEEE/ACM Transactions on Networking, vol. 8, pp. 479-492, Aug. 2000.

[Bau02] Florian Baumgartner. Quality of service support by active networks, February 2002. PhD Thesis, RVS Group, Institut of Computer Science and Applied Mathematics, University of Bern, Switzerland.

[BBa] Florian Baumgartner and Torsten Braun. Distributed emulation of IP networks. Speedup Workshop, Bern, March 22-23, 2001.

[BBb] Florian Baumgartner and Torsten Braun. Quality of service and active networking on virtual router topologies. In Hiroshi Yasuda, editor, Active Networks, Second International Working Conference, IWAN, Lecture Notes in Computer Science, pages 211-224, Tokio, Japan, October 2000. Springer. ISBN 3-540-41179-8.

[BBc] Florian Baumgartner and Torsten Braun. Virtual routers: A novel approach for qos performance evaluation. In Jon Crowcroft, James Roberts, and Smirnov Mikhail, editors, Quality of Future Internet Services, First COST 263 International Workshop. QofIS, Lecture Notes in Computer Science, pages 336 to 347, Berlin, Germany, September 2000, Springer. ISBN 3-540-41076-7.

[BBKW] Florian Baumgartner, Torsten Braun, Evelin Kurt, and Attila Weyland. Virtual routers: A tool for networking research and education. Computer Communications Review Vol. 33 No. 3, pp. 127-135, ISSN: 0146-4833, July 2003.

[Bec02] Kent Beck. *Test Driven Development: By Example*. Addison-Wesley Pub Co, 1st edition, November 2002.

[BFI+] R. Boivie, N. Feldman, Y. Imai, W. Livens, D. Ooms, and O. Paridaens. Explicit multicast (xcast) basic specification. Internet Draft draft-oomsxcast-basic-spec-01.txt, March 2001. Work in progress.

[BGB] Roland Balmer, Manuel Günter, and Torsten Braun. Video streaming in a DiffServ/IP multicast network. Workshop Advanced Internet Charging and QoS Technology at Informatik 2001 (ICQT), Vienna, September 26-29, 2001.

[BGB+00] Gregory Bollella, James Gosling, Benjamin Brosgol, James Gosling, Peter Dibble, Steve Furr, and Mark Turnbull. *The Real-Time Specification for Java*. Addison-Wesley Pub Co, 1st edition, January 2000.

[BL] Torsten Braun and Linqing Liu. Multicast for small conferences. 6th IEEE Symposium on Computers and Communications (ISCC 2001), Hammamet, Tunesia, July 3-5, 2001.

[BLBF] Robert Braden, Bob Lindell, Steven Berson, and Ted Faber. The ASP EE: An active network execution environment. Document can be found at: http://www.isi.edu/~faber/pubs/DANCE.ARP.FINAL.pdf, visited on February 17th 2004.

[Bro00] Marc Brogle. Active networking mit ANTS. Computer science project for the RVS group, Institut of Computer Science and Applied Mathematics, University of Bern, Switzerland, May 2000.

[Bro01] Ian Brown. End-to-end security in active networks, September 2001. PhD thesis, Department of Computer Science, University College, London (GB), Webpage can be found at: http://www.cs.ucl.ac.uk/staff/I.Brown/pimms/thesis.pdf, visited on February 17th 2004.

[Cas] Kenneth Castelino. 3DES and encryption. Webpage can be found at: http://kingkong.me.berkeley.edu/~kenneth/courses/sims250/des.html, visited on February 17th 2004.

[dMCPT] H. de Meer, A. La Corte, A. Puliafito, and O. Tomarchio. Programmable agents for flexible qos management in ip networks. IEEE Journal of Selected Areas in Communication, 18(2), February 2000.

[GB] Erich Gamma and Kent Beck. Official junit web page. Webpage can be found at: http://www.junit.org/index.htm, visited on February 17th 2004.

[GBBa] M. Günter, M. Brogle, and T. Braun. Secure communication: A new application for active networks. International Conference on Networking (ICN'01), Colmar, France, July 9-13, 2001.

[GBBb] M. Günter, M. Brogle, and T. Braun. Secure communication with active networks. Technical Report IAM-00-007, IAM, 2000. Webpage can be found at: http://www.iam.unibe.ch/˜rvs/publications/, visited on February 17th 2004.

[GKPR] R. Guerin, S. Kamat, V. Peris, and R. Rajan. Scalable QoS provision through buffer management. Proceedings of the ACM SIGCOMM '98 conference on Applications, technologies, architectures, and protocols for computer communication, Vancouver, British Columbia, Canada, Pages: 29 - 40, 1998, ISSN:0146-4833.

[Gul02] Ceki Gulcu. Short introduction to log4j, 2002. Webpage can be found at: http://logging.apache.org/log4j/docs/manual.html, visited on February 17th 2004.

[Gul03] Ceki Gulcu. *The complete log4j manual*. QOS.CH, February 2003.

[Gün01] Manuel Günter. Management of multi-provider internet services with software agents, 2001. PhD Thesis, RVS Group, Institut of Computer Science and Applied Mathematics, University of Bern, Switzerland.

[Hua02] I-Hsuan Huang. Active networks: An overview. Document can be found at: http://www.din.uem.br/˜ra/artigos/20020612.pdf, visited on February 17th 2004, 2002.

[IEE97] IEEE P1363 Working draft, appendices, 1997.

[Kue] Geoff Kuenning. International Ispell. Webpage can be found at: http://fmg-www.cs.ucla.edu/fmg-members/geoff/ispell.html, visited on February 17th 2004.

[Lia99] Sheng Liang. *The Java Native Interface: Programmer's Guide and Specification*. Addison-Wesley Pub Co, 1st edition, June 1999.

[mic] Sun microsystems. Official java web page. Webpage can be found at: http://java.sun.com, visited on February 17th 2004.

[PR02] Samuele Pedroni and Noel Rappin. *Jython Essentials*. O'Reilly, 1st edition, March 2002.

[Rar] Rarlab. Official rarlab web page. Webpage can be found at: http://www.rarlabs.com, visited on February 17th 2004.

[RSA77] R. L. Rivest, A. Shamir, and L. M. Adelman. A METHOD FOR OBTAINING DIGITAL SIGNATURES AND PUBLIC-KEY CRYPTOSYSTEMS. Technical Report MIT/LCS/TM-82, MIT, 1977.

[THL01] Patrick Tullmann, Mike Hibler, and Jay Lepreau. JANOS (java-oriented active network OS). Appears in IEEE Journal on Selected Areas of Communication. Volume 19, Number 3, March 2001.

[Tiw00] Manish Tiwari. Active networks, 2000. Document can be found at: http://www.cse.iitk.ac.in/~dheeraj/reports/active.pdf, visited on February 17th 2004.

[TSS+97] D. Tennenhouse, M. Smith, W. Sincoskie, D. Wetherall, and G. Minden. A survey of active network research. IEEE Communications Magazine, 35(1):pages 80 to 86, January 1997.

[Weter] D. Wetherall. ANTS - An Active Node Transfer System, 1997 December. Webpage can be found at: http://www.sds.lcs.mit.edu/activeware/ants/, visited on February 17th 2004.

[WGT98] D. Wetherall, J. Guttag, and D. Tennenhouse. ANTS: A toolkit for building and dynamically deploying network protocols, April 1998. In IEEE OPENARCH '98, April 1998. San Francisco.

[WPA] ANEP: Active Network Encapsulation Protocol. Webpage can be found at: http://www.cis.upenn.edu/~switchware/ANEP/, visited on February 17th 2004.

[YNR98] Ikjun Yeom, Narasimha, and A.L. Reddy. Realizing throughput guarantees in differentiated services networks, November 1998. Technical report, Texas A & M University.