Lessons Learned on Enhancing Performance of Networking Applications by IP Tunneling through Active Networks

Tomas Koutny and Jakub Sykora
Faculty of Applied Sciences
University of West Bohemia
Plzen, Czech Republic
txkoutny@kiv.zcu.cz, jsykora@students.zcu.cz

***Abstract*— In 1995, DARPA initiated a work on a programmable concept of computer networking that would overcome shortcomings of the Internet Protocol. In this concept, each packet is associated with a program code that defines packet's behavior. The code defines available network services and protocols. The concept has been called Active Networks. The research of Active Networks nearly stopped as DARPA ceased funding of research projects. Because we are interested in research of possible successors to the Internet Protocol, we continued the research. In this paper, we present an active network node called Smart Active Node. Particularly, this paper focuses on its ability to translate data flow transparently between IP network and active network to further improve performance of IP applications. We describe the translation mechanism, its possible use and discuss particular implementation aspects.**

Keywords- Active Networks, Smart Active Node, IP tunneling, routing

I. INTRODUCTION

This paper extends the original paper [1] (Sections 1 – 6, sub-sections A and B of Section 10 and a portion of Section 11), as it captures recent advances on the project since Section 5, sub-section D.

Today, IP networks suffer from low scalability and deployment of new networking services is a subject to a long standardization process. A particular problem that lies within the scope of this paper is content delivery over IP, with respect to time-sensitive traffic – e.g video. Simply said, an effective solution is possible with a programmable network and for that task we need Active Networks [2, 3].

For example, a number of multi-cast schemes and protocols were developed. They try to do their best in optimizing a multi-cast tree to satisfy and guarantee a proper quality of service. These protocols cover multi-cast tree creation, optimization and client group membership management. This requires special hardware and software support from both network and clients. In fact, there is a complex overlay network built on a top of the IP network. While it addresses needs of today, there is still a room for an improvement [4]. We desire to be ready even for needs of tomorrow.

We do not aim at solving a particular problem. We try to build a general solution, which could be used to solve a variety of tasks and issues in a simple manner. To solve this general problem, we did not decide to use a traditional network. Instead, we decided to use the concept that is known as Active Networks.

Active Networks is such concept, where every network node is active, when compared to passive elements used today. The activity is meant as the ability of a network node to process data in a context of application that created them. To make this possible, a packet has been superseded with a capsule. Along data, each capsule is associated with a reference to a program code. The code is downloaded through the network as needed and executed, as a capsule is run at a node. As the code executes, the node is able to handle the capsule's data in an application specific context. Thus, it is possible to teach the network new things on the fly. Note that capsule can route itself.

Active application is such networking application that injects capsules, which replace packets, into the network. In turn, a capsule may inject another capsule or an active application into the network. Both, application and capsule have an access to a server-offered API to use its functionality. Any custom code runs in a sand-box that is called Execution Environment.

As it is not realistic to assume that Active Networks would suddenly replace IP networks, these two networks would have to co-exist for a certain period. Thus, instead of awaiting a revolution in networking, we focus on adding more functionality to existing IP solutions via tunneling them into the world of Active Networks.

A preceding work is presented in Section 2. Section 3 explains our motivation. Fourth section describes proposed solution, while the next section is focused on implementation. Sections 6 and 7 focus on policy-based routing and worth-path routing. We discuss results in Section 8. The following section gives additional details on the most needed improvement – code execution. Related work is given in Section 10. Section 11 finishes with conclusion.

II. PRECEDING WORK

The PANDA [5] project was the proof of the concept of tunneling the IP protocol over an existing active network. The PANDA software ran on a top of ANTS [6] active-network server. It was a demonstration of active network's capability to transfer UDP datagrams transparently and to possibly recode contained video stream in order to satisfy bandwidth limits. The project, namely its PIC component, was implemented as a kernel module that communicated through a BSD socket with the active network node. The node performed recoding and distribution of the stream. The demonstration showed that there is no need to modify neither source stream server nor the client software. By using active network as the underlying network, there was a significant spare of bandwidth and better QoS, was presented; QoS stands for Quality of Service.

III. MOTIVATION

In our research, we focused on problems, which showed up in the preceding work. They include IP tunneling, security, resource allocation and performance. We develop our own, general-purpose active network server. Its design addresses many shortcomings of the previous active network implementations. Preceding projects were generally aimed at particular problems, but without researching consequences among those. Capsule and application programming interfaces, performance and security have to be addressed altogether, not as standalone issues.

An important issue of active networking is performance. This is given by a number of possibly flowing capsules, and the need to execute their code in a sand-boxed environment to guarantee a required security. This is very challenging goal and no satisfying solution was present. Perhaps, this was the main reason, why DARPA ceased research funding on Active Networks.

However, thanks to our research ideas and comparison with other projects, we consider this issue as solvable. Therefore, we did not decide to favor performance over security and server's design.

Thus, not taking the performance as a limiting factor, we have a general-purpose active server that anyone can deploy, write an application and investigate its behavior without studying server's source code.

The research project is called Smart Active Node, SAN in short [7].

IV. PROPOSED SOLUTION

Our efforts on building an active network started with an idea of a general-purpose server and IP-tunneling.

A. *Generality, Usability and Security*

The server does not make any assumption about applications, which will run in the network. However, programmer of an active application should aim for low resources consumption. Otherwise, security monitor may consider increased demands for resources as a possible attempt of a denial of service attack, or a malfunctioning application. The resource is anything that can be allocated to the application, or a capsule – i.e. memory, processor time, network bandwidth, etc.

Developing an active application should be as comfortable as developing a traditional application. Usage of a common IDE to develop active application is desired.

Any active code runs in a sand-boxed environment to meet security measures. No instance of any program code can affect another instance by mistake. For an inter-process communication, it is necessary to use server-offered API.

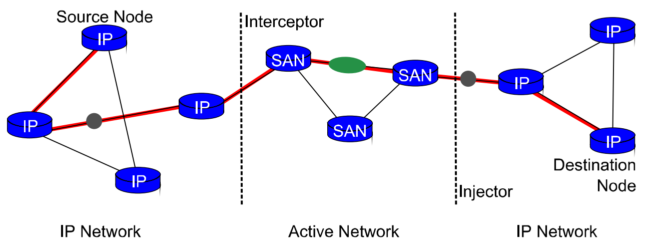

Figure 1. IP Tunneling Network Scenario.

In the present implementation, programmer supplies Java byte-code that is executed in an execution environment, the sand box, and controlled by the security monitor.

B. *IP Tunneling*

The goal is to let the Smart Active Node to provide a seamless IP tunneling through the active network. Fig. 1 depicts an illustrative network scenario. Consider two IP networks interconnected with an active network, where a source node sends IP packets to a destination node. The active nodes, which are connected to the IP networks, act as hybrid devices with both, IP stack and active networking functionality. As the IP packet gets to the hybrid node, it is intercepted at the third ISO/OSI layer. A component named Interceptor is responsible for this.

Then, the packet is encapsulated into a capsule and routed through the active network to the hybrid border node that is connected to the destination IP network. It is the capsule's program code, what makes the difference in performance. Note that as SAN runs on a standard operating system, both networks can overlay each other as well.

The destination-border hybrid node unpacks capsule's payload and injects the extracted packet into the IP network. The responsible component is called Injector.

Finally, IP network routes the packet to its IP destination.

The principle is the same for both directions so that Interceptor-Injector pair is present on each border node to satisfy two-way communication.

Fig. 2 depicts a view on assignment of responsibilities. Active network server and IP stack of underlying operating system cooperate. Oriented lines show the data flow. Starting with data coming through the IP stack, the interceptor component, called saninterceptor, receives the data as a

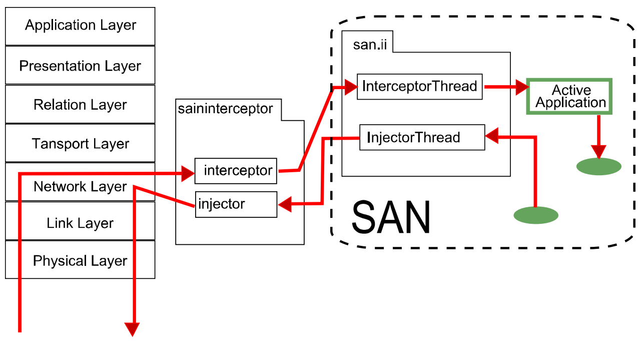

Figure 2. Linux IP Tunneling Components.

packet. Subsequently, it is passed to the ii component; ii stands for interceptor-injector. This component is responsible for encapsulating it into a capsule.

When receiving data as a capsule from an active network, the ii component delivers the data to the injector component. Then, the data are injected at the ISO/OSI network layer into the IP stack.

V. Implementation

Having the design, we continued with implementation. Initially, we assumed that Java and JVM will be fast enough, to get acceptable values of throughput and latency. Therefore, only OS-dependent parts of the tunneling were written in C++.

A. *Generality, Usability and Security*

As our solution is a general-purpose server, there is no special software needed to create custom active applications. Recently, some basic applications already work and development of others is in progress. The working ones include ping, trace route and IP tunneling. The work in progress comprises of dynamic routing, telnet, SSH and possibly a port of AVNMP [8] – a tool to predict network's load.

SAN active application is written in the standard Java language, compiled and packaged into a Java archive with a manifest file. The applications can be developed in IDE such as Eclipse or Netbeans with no expenses.

The application's, or capsule's, code is interpreted as a Java byte-code in the present state. We developed our own byte-code interpreter. As it has been written from scratch to allow strict control over the execution process, it interprets everything, down to Java native methods. As the result, nearly every valid Java construct can be used to create an active application. Moreover, we have a full control over the code. Thus, passing a special file system identifier to obtain undesired access on particular operating system can be forbidden, as well as a simple constructs like calling System.exit(0) to shut down the server maliciously. Preceding works, such as ANTS, used directly the Java machine they run within, thus virtually providing no security.

B. *Optimization*

SAN started as a Java project for various reasons. As already mentioned, we need to address the performance. To improve it, a C++ clone of the server is being written. From this step, we expect a performance increase and the possibility to deploy the server on such nodes, where Java is not available, e.g. switches and routers.

In an active network server, the most likely bottleneck is byte-code interpreter and scheduler. To run the byte-code, it is necessary to prepare execution environment, i.e. the sand box, and to schedule it for execution. Preparing the execution environment is a time-significant part of total run-time, in a case of shortly running capsule codes such as ping. Thus, the overhead does not matter, if the application run-time is long and frequency of runs is low. However, it matters with applications such as the IP tunneling. The IP tunneling run-time per capsule is very short and the frequency of runs can be very high. It depends on the data stream being transferred.

To speed up the code execution, we would like to have a mechanism that would optimize parts of code being executed frequently, and to cache them subsequently. The optimization would be a byte-code transformation into processor's native instruction set.

Last optimization task is to examine the internal scheduler. It is currently implemented as a fair-share.

C. *IP Tunneling on Linux*

We have implemented the IP tunneling over active network on Linux first.

The idea behind the tunneling is following. If we want to pass IP packets transparently through the active network, we have to intercept IP packets either on physical layer, link layer or network layer to prevent the operating system from managing these packets. Otherwise, the operating system could possibly send ICMP error packets back, because it is not aware of being a part of active network.

We chose to use unmodified Linux kernel along with the Netfilter/Iptables [9] project to preserve simplicity, generality and ease of use. We used the Iptables' NFQUEUE target along with the ipq library for queuing packets into user space. There is a benefit coming from the usage of this approach – we can easily decide, which packets from and to the IP networks are transferred through the active network.

After en-queuing a packet, entire datagram containing all headers is fetched into user space with libipq API calls. And, it is sent unmodified through a network socket to the SAN along with information about active code that handles its data.

Packet data and meta information exchange between saninterceptor and SAN ii component is accomplished through a standard socket, while using a special type of PDU to transfer the data. The PDU format and primitive data types are shown in Fig. 3; PDU stands for Protocol Data Unit. The first position of the PDU is the name of the active application being executed upon receiving the data. Then, an array of

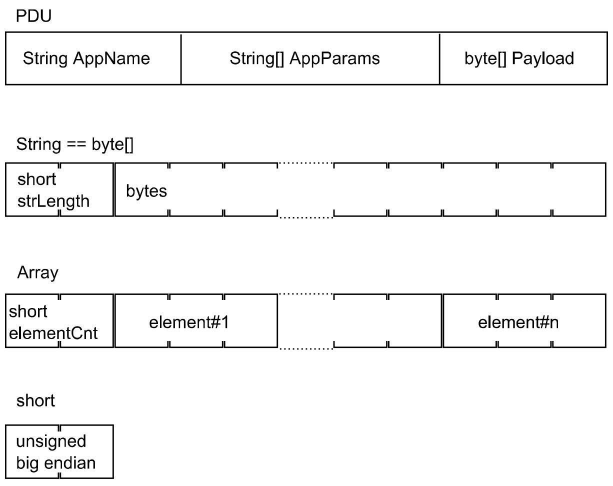

Figure 3. SAN Interceptor-Injector PDU.

application parameters apply. For example, they can represent routing, QoS or ToS information; ToS stands for Type of Service. Finally, entire datagram is attached. The PDU is flexible enough to handle a datagram up to 65kB. This is enough even for super jumbo frames.

Upon receiving PDU, the active node passes the datagram to the active application that is responsible for the IP tunneling. The application creates a capsule and injects it into the network. When the capsule arrives at the destination active node, datagram is unpacked and sent through the socket to the injecting application. Injector injects the datagram into the IP network. As the packet is not modified on its route, the process is fully transparent to IP applications.

We did tests with HTTP, SSH and FTP protocols. They worked flawlessly, like if no active network was presented.

D. *IP Tunneling on Windows*

We continued with implementation of the WDM driver model that applies to Windows 2003, Vista and 7. ReactOS uses the WDM model as well, but we made no tests on ReactOS yet. Fig. 4 depicts the implementation.

Legacy IP applications communicate via the TCP/IP NDIS protocol as usually. SAN filter intercepts the communication. Intercepted IP packet is accompanied with additional information and sent to SAN server via an inter-process communication. In SAN address space, an active application converts it into a capsule. Then, SAN server handles the capsule in a standard way.

When the capsule is to be converted back into the IP packet, NDIS driver does this as instructed by SAN. A legacy IP application gets the packet from the TCP/IP NDIS protocol as usually.

With the further development, we aim to support two kinds of applications – legacy IP applications and SAN-aware applications. SAN-aware applications would be free to use SAN capabilities directly. Thus, they would be able to exercise a finer control over the transmission.

VI. WORSE-PATH ROUTING

References [4, 5] give existing enhancements on multicast and tunneling of existing IP applications. We would like to go a step further by proposing such routing scheme that will rearrange network flows to benefit time-sensitive networking applications.

A. *Policy-Based Routing*

Let us classify network traffic into two categories. First one is time-sensitive traffic, for instance IPTV and VoIP. Second category is such traffic, where it is possible to tolerate some increase of delivery delay. For instance, SMTP and file-sharing services fall into this category.

We do not use terms real-time and non-real time traffic, because we discuss additional scenarios such as MPI in subsection D. While we assume a possible benefit for MPI, we do not assume a real-time application using MPI.

Multiple routes to target nodes may exist in a computer network, or an interconnection of computer networks – especially the Internet. Some routes are better in terms of

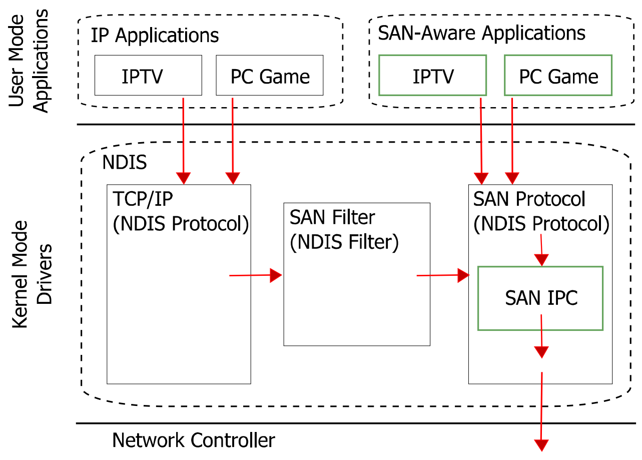

Figure 4. Windows IP Tunneling Components.

bandwidth, load, reliability, number of hops, etc. Routing algorithms try to find an optimum route. Regarding Internet Service Providers (ISPs), a price of link plays a role as well.

Let us consider an ISP with two different links to other ISP. One link is cheaper, but there is a lower bandwidth. To reduce costs, ISP would prefer such policy that would route most of the traffic through the cheaper link. Nevertheless, ISP should route the time-sensitive traffic through the faster link to maintain a quality of services to customers. In IP, this concept is implemented as policy-based routing.

However, the other ISP may not be interested in maintaining such quality of services to the customers of the traffic-originating ISP. By addressing this issue, the proposed concept differs from policy-based routing, as it is implemented in IP.

We give such IP tunneling scheme that routes the delay-tolerant traffic through slower links. As a result, it reduces the need to throttle the transmission speed of time-sensitive traffic on faster links. The proposed approach does not impose a need for agreement on common routing policies between two ISPs.

B. *Principle*

First task is to intercept such IP packets, which do not belong to the time-sensitive traffic. Then, we wrap these packets into capsules. Finally, associated program code routes the capsules through slower links – the worse-path.

Let us consider SMTP and IPTV for demonstration. Once SMTP server retrieves MX record for target domain, it opens a TCP connection to the destination server. Routers will direct the flow of connection's packets according to routing tables, as it would happen with the IPTV packets. SMTP and IPTV packets may share the same link. QoS can throttle transmission speed to favor time-sensitive traffic such as IPTV. However, QoS cannot route a particular TCP connection over a different link to gain yet more bandwidth for the IPTV. With IP and policy-based routing, we would need ISPs, which agreed on compatible routing policies. With a programmable network, we can apply the following concept.

First, we need an additional routing table at the router. To fill the table, it is necessary to modify routing metric so that it favors slower links. For example, OSPF uses inverse value of bandwidth. Then, we would take the bandwidth as the metric.

Second, we need a data unit that would be routed by the alternative routing table. In active networks, the router would execute the capsule's code. So, the capsule would look-up the alternative routing table and set its destination accordingly. If the router would not execute the code, e.g. for security reasons, the capsule would be forwarded according to the standard routing table. So, it would reach the destination as well, just sooner.

Capsules route themselves through slower links. Thus, they leave more bandwidth for the time-sensitive traffic on the faster links. Considering a possibility of different routing policies in transit networks, capsule's behavior increases the probability that time-sensitive traffic will use the faster links.

As capsule's program code does not change, the capsule acts the same way in all transit networks. Therefore, no two ISPs have to make a prior agreement on common routing policies.

C. IP-Programmable Hybrid Network

Let us consider a scenario, where a programmable node would aid a traditional IP network. As we do not tunnel the time-sensitive traffic, we can route it the standard way. On the other hand, the tunneled traffic is wrapped into capsules. We can distinguish such traffic easily, e.g. by port number, or a header bit. Therefore, it is possible to establish an efficient routing policy. Such policy would route capsules to the programmable node, while leaving rest of the traffic untouched.

Fig. 5 depicts a case scenario. Various clients from the source network #1 want to connect to particular hosts in the destination network #4. There is a policy-based routing enabled at the router that acts as their default gateway. It identifies particular protocols by port numbers. Selected traffic goes to the SAN server. Otherwise, the router forwards rest of the traffic to the IP-based border router. SAN server intercepts incoming IP packets and transforms them into capsules. According to programmable rules, it forwards them to the next SAN server. Note that the associated code can do much more than just policy-based routing. SAN servers in the transient networks act the same way. In the destination network #4, SAN server transforms capsules back into IP packets and forwards them with the standard IP routing mechanism.

The IP-programmable hybrid is not a fully programmable network. However, Fig. 5 depicts such scenario, where it is possible to route a defined amount of traffic to the programmable servers. As a result, we can test responsiveness and stability of the programmable servers to given load, while having backup routes.

Note that it is not necessary to deploy the hybrid network at the Internet scale. It can serve as well for networks of a single organization, or its department.

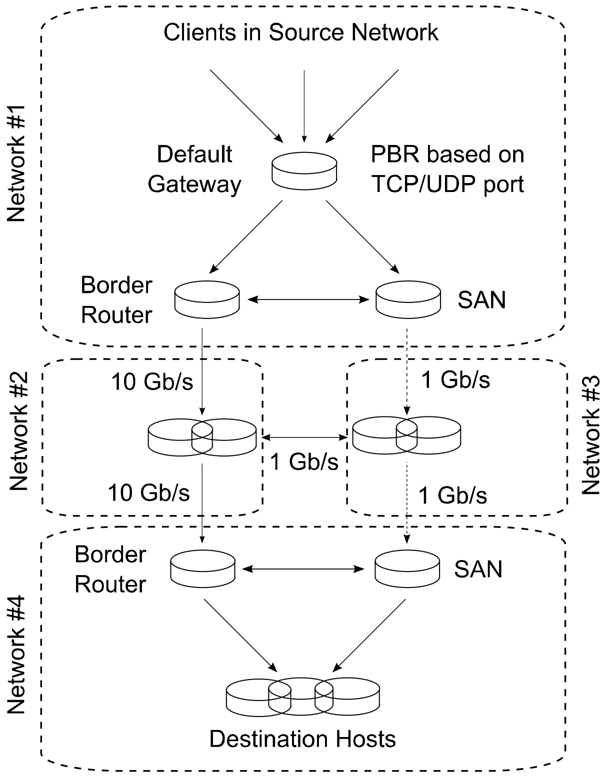

Figure 5. Hybrid Scenario of Worse-Path Routing.

D. Additional Case Scenarios

Let us consider a grid computing, for an illustrative example. A large grid may consist of several sub-grids, which are connected with slower links than the links inside the sub-grids. For distributed computing, there are tools such as GridMPI and PVM available. These libraries provide means for asynchronous and blocking communication.

For example, MPI_Send function is blocking. The caller does not continue its execution, until it receives a confirmation message. The communication overhead affects caller's performance, i.e. the completion time. For this reason, we can consider such communication as time-sensitive. Therefore, we should route it through faster links.

On the other hand, MPI_Ibsend function is non-blocking. The caller continues its execution, while MPI delivers the message. For such programs, we could route such messages through slower links to reduce the waiting time of blocking operations such as MPI_Send.

Another possible scenario is secure, anonymous communication. The TOR project provides a network of nodes, which route communication in such manner that it is too hard to find its origin. TOR uses so-called onion routing and it supports applications, which use TCP. With tunneling, SAN could implement the same behavior for any packets, i.e. to build a secure network by default. With policy-based routing and client's IP, it would be possible to enable such service per individual user.

E. *Comparison with Source Routing*

IP offers loose and strict source routing. With such routing, packet's route is set in advance by the source node. While the strict routing sets entire path, routers may forward the packet through other routers as well with the loose routing. In both cases, sender must have such knowledge of topology so that the packet reaches the destination. There are two problems.

First, internal topology of other ISP is supposed to be opaque. At the best, there is no guarantee on knowledge of the topology, including bandwidths, utilization and other factors. In a worse case, ISP can choose to block the source routing.

Second, we do not discuss a use of source routing for an administrative task. We discuss use of the source routing for a regular traffic. In such case, the source node would have to maintain a complete routing table for entire Internet. This would impose overwhelming requirements on the node, thus rendering such solution as impossible.

In contrast to the source routing, SAN-based solution uses the well-established concept of routing tables along the route to the destination node.

VII. Experimental Implementation of Worse-Path Routing

To implement the worse-path routing, we use the following technologies – AntNet for routing and Rendez-Vous to expose an additional programming interface to active code of capsules.

A. *AntNet*

First, we needed a routing algorithm. To benefit from the programmability, we implemented the AntNet algorithm [10]. This algorithm was originally designed for mobile agents. It is inspired by a behavior of ant colony. Using indirect information, a simulated pheromone trail, simulated ants find shortest path thanks to their cooperative behavior.

Making a reference comparison to OSPF, AntNet has better distribution of packet delays and negligible impact on the use of network bandwidth [10]. There are recent improvements to AntNet. They further improve throughput, stability and shorten time delay [11, 12]. Improved AntNet deals with topology changes better than OSPF does [11]. Reference [10] gives a comparison with other routing protocols as well.

For our experiments, we simulate Internet with a set of interconnected nodes. We simulate networks of individual ISPs with these nodes. In this fashion, we use AntNet as an exterior gateway routing protocol.

B. *Rendez-Vous*

Having the routing algorithm implemented, we needed to implement an alternative routing table in a general fashion. We wanted to avoid any ad-hoc solution. We implemented Ada rendez-vous synchronization mechanism into our Java byte-code interpreter.

In our implementation, an active application can register a rendez-vous server. Incoming capsule can lookup a particular server and call its entries. This way, we have achieved a possibility of having so-called installable APIs. As a result, SAN exposes just the minimum set of functions through a pre-defined interface. Then, it is upon networking services and applications to expose desired application programming interfaces – APIs. It is possible to register and unregister particular API dynamically, without a need to restart SAN server.

The choice of rendez-vous has a security background. Synchronizing with a monitor, the calling thread executes monitor's code, thus using its internal data structures. A malicious code could possibly exploit such design. With rendez-vous, the calling thread is suspended, until the called thread, the server, finished the execution. Thus, the caller has no access to server's internal data structures by the very principle of rendez-vous. This is important to us as we consider a possibility of execution environment reuse to speed-up code execution.

There are stability benefits as well. Having a rendez-vous executing in a standalone thread, we avoid priority problems. As the rendez-vous thread keeps its priority, a critical section will not be blocked for too long by a thread with low priority. Moreover, the rendez-vous thread can exercise a better control over resources being allocated in a critical section, than calling threads could do. An unknown calling thread is more likely to be terminated by security monitor than a rendez-vous server thread is.

Java has no native program construct to implement rendez-vous as Ada has. There are two ways to implement the mechanism into present Java language standard.

One way is to develop a pre-processor that would allow Ada syntax in .java file. The preprocessor would generate a .java file that a Java compiler would accept. Therefore, any debug info would be valid for the generated .java file.

It is always necessary to develop a package, which would synchronize threads in the rendez-vous manner. Therefore, the second way is to use this package directly, without the pre-processor. This way, we have a .java file that programmer understands, compiler can process it and generates a debug info for it. We chose this way.

The following code shows a fragment of the rendez-vous server, which implements the worse-path routing table.

```
public void worsePathRoutingServer() {

//1. Register Rendez-Vous server
if (!applicationAPI.registerServer(this,
                                WPRTable_GUID)) {
  //Error registering the server, however
  //traffic is still routable. Capsules will
  //just use the default routing table.
  return;
}

//2. Register supported entry-calls
applicationAPI.registerEntryCall(WPRTable_GUID,
                                GetGateway_NAME);

//3. Handle the entry-calls
try {
  applicationAPI.makeAccept(WPRTable_GUID);
} catch (InterruptedException ex) {
  handleException(ex);
}
//4. Clean-up
applicationAPI.unregisterServer(WPRTable_GUID);
} //end of worsePathRoutingServer()
```

The following code fragment shows as capsule routes itself, using the worse-path routing table:

```
public void routeCapsule() {

//1. Get capsule's destination; it is not stored
//   in capsule's header as the header may
//   change on the route. Therefore, we store
//   the real destination in the payload.

NetIdentifier realDst = readCapsuleDestination(
                       capsuleAPI.getPayload());

//2. Try to get worse-path routing record
//   for the real destination.
try {
  RoutingRecord rr =
       capsuleAPI.callEntryCallByName(
       WPRTable_GUID, GetGateway_NAME, realDst);

   //3. If the previous call succeeded, extract
   //   the gateway and set it as capsule's
   //   destination.

  if (rr != null)
    capsuleAPI.setDestination(rr.getGateway)

   //4. In a case of failure, set the real
   //   destination. Then, the capsule will
   //   reach its destination through a normal
   //   route. However, this code will attempt
   //   to resume the worse-path routing
   //   at the very next node.

   else capsuleAPI.setDestination(realDst);


} catch (InterruptedException ex) {
  capsuleAPI.setDestination(realDst);
}

} //end of routeCapsule()
```

While we have the key components done, the worse-path routing is not finished yet. As given in the following section, we need to switch to the SAN C++ port, first. Then, we can finish the implementation with such execution speed that is fast enough for a productive use. As SAN servers make a distributed environment, where each node is a parallel application, the execution speed is an important factor for debugging.

VIII. Results

Initially, we expected better performance results than we achieved. Eventually, it turned out that speed and security with Java-in-Java in JVM will not run fast enough, despite source code optimization, runtime profiling and performance tuning done by JVM. For this reason, the paper ends with a focus on speed of code execution.

A. Background

The PANDA project [5] was built on a top of ANTS project [6]. ANTS project ran in Sun JVM. The JVM executed capsule code as well as ANTS' code. While this allowed a greater throughput than a Java-in-Java approach, the solution was not secure enough. Later on, secured solutions appeared. They included PLAN [13], RCANE [14], SANE [15] and SNAP [16]. They were able to verify integrity of server's code and configuration, and authenticity of capsule's code. Also, some of them limited programming constructs to avoid creating of a possibly dangerous program code.

We decided to strengthen the security measures by being able to monitor code execution on-the-fly. For this reason, we replaced the use of Sun JVM with our own Java-in-Java interpreter. This became the performance bottleneck of our server. Eventually, it became obvious that we need to abandon Sun JVM to run the server. We chose to port the server to C++, while leaving the Java development branch as a sand box. Presently, we use it to test new ideas and to develop active protocols in advance, prior to finishing the C++ port of the SAN server.

B. Linux IP Tunneling

We performed a couple of tests with the IP tunneling implementation. In all tests, there was a saturating traffic flow from one computer to another. We generated a continuous stream of IP packets to saturate link's bandwidth.

The first test was the performance test of the saninterceptor itself. It was aimed to prove correct memory management, effective CPU and bandwidth usage. Table 1 shows results for two directly connected PCs with 100 Mbps network cards and the same PCs connected through two instances of saninterceptor. Looking at the Table 1, we can say that use of saninterceptor nearly does not affect data transmission, even if it is implemented by simple means. N/A means that values were not observable or affected at all.

The second test was the performance test of a complete system, i.e. including SAN, active applications, etc. This test revealed some drawbacks in SAN's implementation. They are related to running many instances of short-run-time applications and capsules.

Although the tunneling results were not satisfying, they showed that the IP tunneling works and that it can be used for tunneling of applications like HTTP and SSH. Nevertheless, it became clear that we need to improve byte-code execution prior making any other substantial changes.

TABLE I. Initial Saninterceptor Performance Results on Linux to Linux

	CPU	Memory	Latency	Throughput
direct connection	N/A	N/A	<2ms	94 Mbps
Saninterceptor only	N/A	18kB	2ms	90 Mbps
SAN + saninterceptor	100%	N/A	>200ms	120 kbps

TABLE III. CODE EXECUTION TIMES ON WINDOWS

Environment	Average Time [sec]	First Time [sec]
SAN Java-in-Java	>> 1	>>1
SAN C++ JVM	1.37200	1.38000
1st Sun JVM 1.6.0_17 64-bit	0.02350	0.02594
1.6.0_17 64-bit -Xcomp	0.02535	0.02605
2nd Sun JVM 1.6.0_17 64-bit	0.01255	0.01295
1.6.0_17 64-bit -Xint	0.22561	0.22412
1.6.0_17 64-bit –g:none	0.01301	0.01363
VC2008 x64 Debug	0.07000	0.07000
VC2008 x64 Release	<0.00001 0.01000 occassionally	<0.00001

C. Execution Environment Benchmark

To compare performance of particular environments, and to estimate a minimum needed performance, we created a benchmark test. Since the C++ port is not finished yet, we cannot evaluate network-specific operations. Therefore, the benchmark computes a matrix determinant in such manner, that is uses memory, ALU and floating point instructions.

Table II gives results for Windows 7. To eliminate side effects of operating system, such as caching and program loading at random addresses, we ran the test for 10 times. Then, we ran the test for another 30 times and computed average execution time. The First Time column gives time just for the very first run. As the server has to execute a program code for a first time, as well as it may execute a particular code frequently, we give both – average and the first time. We collected the presented results on Intel64, family 6, model 23, stepping 10, frequency 2.40 GHz.

Table III gives results for Debian 4.3.2-1.1. The machine is part of Czech National Grid Project – MetaCentrum. This affects the software equipment. It runs on Intel Xeon, family 15, model 4, stepping 3, frequency 2.80 GHz. The benchmarking procedure was the same.

Using Sun JVM, we performed tests with non-standard switches. After the regular test, we run the test again, but with the non-standard –Xcomp switch. According to the documentation, everything should be optimized. Program loading took ~1 second on Windows, ~3 seconds on Linux. On Windows, the execution time did not change. On Linux, the execution time was longer. Then, we run the JVM again, but without the switch. On Linux, the execution time returned back to normal. On Windows, it was reduced by ~50%. Perhaps, there is some caching effect that causes such behavior.

Beside –Xcomp, we tested the –Xint switch as well. Accordingly to the documentation, the code should be interpreted only.

For comparison, we ran a C++ port of the benchmark with VisualC++ 2008 64-bit compiler, and with GCC 4.3.2 32-bit compiler.

SAN C++ JVM does not compile on Linux yet, so it is not included in Table III.

TABLE II. CODE EXECUTION TIMES ON LINUX

Environment	Average Time [sec]	First Time [sec]
SAN Java-in-Java	>> 1	>>1
SAN C++ JVM	N/A	N/A
1st Sun JVM 1.5.0_10 32-bit	0.06137	0.06185
1.5.0_10 32-bit -Xcomp	0.09401	0.09388
2nd Sun JVM 1.5.0_10 32-bit	0.06127	0.06138
1.5.0_10 32-bit -Xint	0.46654	0.46329
1.5.0_10 32-bit –g:none	0.06257	0.05964
GCC x86 –O0	0.02000	0.02000
GCC x86 –O3	<0.00001	<0.00001

D. Discussion

SAN's Java-in-Java interpreter is so slow that it has no point to measure the networking performance. With Java-in-Java, we test flow and logic correctness of program code.

Present JVM of SAN C++ performs much better. However, it is still significantly slower than Sun JVM. The performance results indicate that Sun JVM transforms byte-code into the native instruction set, based on some threshold given by code profiling. However, the optimization does not seem to be as good as it could be, on Windows and Linux x86 platforms.

If we would consider that the optimization does not make an intensive use of available processor registers to favor simpler-to-code utilization of stack, then it is a reasonable appeal to us to pursue the byte-code transformation, instead of developing the Java-in-Java byte-code interpreter further.

So, we already started to implement the byte-code transformation to the C++ port. SAN C++ will transform the byte-code, which is being executed frequently, into the native instruction set. Otherwise, it will execute the byte-code inside its JVM. We chose this rule to avoid program-code cache-trashing, and to reflect the very fact that the transformation takes some time as well. The threshold values of "executed frequently" and program-cache size are a subject to future research.

IX. CODE EXECUTION

To execute the active-networking code, we decided to support two code notations – the byte-code and processor-native code. The byte-code will be either interpreted, or transformed into the native instruction set. Any application can be executed in byte-code. For selected operating systems, the code can be supplied in a processor-native instruction set to support critical operations. Such code has to be signed digitally, and the server has to trust explicitly the particular code and the signer.

A well-written program in C has lower memory requirements than a byte-code equivalent. In addition, compilers such as GCC produce much more efficient native-instruction code than JVM does from byte-code. This is our motivation for allowing the possibility to supply the code in native instruction set. We would like to have popular protocols to be handled as most efficiently as possible, with respect to their share in traffic composition.

A. *Byte-Code Transformation*

Instead of transforming byte-code instructions directly into processor-native instructions, we decided to generate C code. The generated code will use pointer arithmetic to manipulate operands of the byte-code instructions, which are stored in the stack. Then, we will rely on a C compiler for optimization.

For a complex byte-code instruction, such as a newarray or monitorenter, we will call a respective C-coded function. This way, we can handle methods such as System.arraycopy, as JVM already does [24].

B. *Security*

To enforce security measures at the program-code level, we need to forbid particular program constructs and to limit memory and processor-time utilization.

When we encounter an instruction such as anewarray, the byte-code interpreter asks security monitor. A C-transformed code will call a function that will do the same. The security monitor checks current memory allocation status and acts accordingly. If the amount of allocated memory is over a given threshold, the request is denied.

As byte-code interpreter runs, it counts number of executed instructions per interval. Scheduler will not plan the process, if it would overcome an imposed limit. Into the C code, we can insert such code blocks, which will check processor-time utilization and yield the processor eventually.

Alternatively, there are SetThreadContext and GetThreadContext functions on Windows. On Linux, there is the ptrace function. With these functions, we can suspend thread execution and get/set its context. Then, there will be no need to include those code blocks into the generated C-code.

A programmer may desire to call a certain, possibly dangerous method like System.exit. In SAN, calls are checked and possibly denied with black-lists. The configuration may look like this fragment:

```
<roles>
 <role name="anonymous"
        resourceProfile="anonymousProfile">
  <permissions>
     <access name="java.lang.System.exit"
             type="method" allow="forbidden"/>
  </permissions>
 </role>
</roles>

<resourceProfiles maxRunningCapsules="20"
                  maxActiveCodes="50">
 <profile name="anonymousProfile"
          priority="onIdle">
  <cpu type="percent" maxValue="5" />
  <memory type="percent" maxValue="5" />
  <bandwidth type="percent" maxValue="5" />
  <activeCodes maxValue="100" />
  <createdCapsules maxValue="5" />
 </profile>
</resourceProfiles>
```

The concept of roles serves as an additional protection. For example, a time protocol cannot modify routing table, and routing protocol cannot set system time. Uncategorized protocols are most restricted with the anonymous role.

X. Related Work

A number of papers were published on Active Networks in earlier years. Also, some recent works are relevant, although they do not address Active Networks exactly.

A. *Google Chrome*

Taking a closer look on the concept of the Google Chrome [17] operating system, we see a resemblance with the active networks concept. There is a simple, underlying operating system that provides hard application isolation – sandboxing [18]. API and the definition of web services define the Execution Environment. Also, it features a security manager that prevents running a malware.

B. *Google Native Client*

Although the Native Client [19] is not primarily designed for a use in active networks, there are ideas valuable to a high performance execution environment.

C. *AntNet QoS*

Another implementation of QoS that is based on a programmable approach is an adaptation of the AntNet routing algorithm for QoS [20]. The implementation was tested in a heterogeneous network as a part of a multimedia transcoding system. As a server receives a request, it generates ants, which search for a best transcoding path and service, based on desired QoS.

By having the AntNet algorithm implemented, we can continue to implement the QoS capability.

D. *Security*

Reference [21] gives an overview on CSANE active network concept, which aims for security and scalability. CSANE goes for cluster processing. It builds on ANTS, JanOS and Linux.

With rendez-vous, we take a preemptive counter-measure to deal with a possibility of attack, which is based on sharing a memory with another process. There was a similar attack, on the HyperThreading platform. One process obtained data of another process, particularly RSA key. We consider principle of this attack to possibly apply to monitor calls. Reference [22] provides attack details and gives suggestions to designers of operating systems.

E. *Performance Testing*

Reference [23] gives a recent, comparative study of JVM benchmarking – Sun JVM and Oracle JRockit. It concludes that JRockit runs usually 19% to 27% faster. Such numbers support the decision to perform byte-code transformation with SAN C++ internal means, instead of switching to another JVM.

F. *Native Code*

Reference [25] presents a load-redistribution method for distributed applications. As a proof of concept, it uses a special active-network server with no security, but utilizing active programs coded in processor-native instruction set. Execution overhead of this approach is insignificant.

XI. Conclusion

Adding program logic to passive IP packets may lead to a significant increase of network's efficiency [4, 5]. A network flow can adapt to current conditions automatically, as its units of transmission traverse the network. This is paid with increased overhead, as there is an executing code.

SAN is a universal active node server that is capable of IP tunneling to enhance performance of IP applications, as well as supporting newly created, active networking applications. Both can be accomplished in standard operating systems, so that no "overnight" revolution is needed to start benefiting from the active-networking concepts.

The related work shows that the concept of active networks is usable for the future – although, not in a way it was supposed to happen originally. With respect to the advances on the original work [1], we can try to evaluate history of active networks' development. Looking back at the history of active networks from the point of view of SAN development, it seems that the magnitude of initial support was driven more by expectations and possibilities, than it was corrected by development costs and requirements.

Active networks appeared with a proof-of-concept that was implemented with Java. While specialized languages appeared for active programs, a vast majority of well-accepted papers on active networking used Java. This gave the impression that Java is a good choice for development of active-networking server. And, we do not agree.

Speed and security were the major disadvantages, which prevented adoption of active networks. Our results suggest that the use of JVM is the cause. JVM cannot compete with an optimized code from a C compiler like GCC. Comparing Java and JVM with a well-written C program, Java loose, when it comes to memory requirements and code execution speed of both, active program and associated security checks. JVM overhead seems to be too great for software like active-networking server.

We still consider Java as a good choice for using the byte-code as active program notation, across different operating systems and processors. Also, Java benefits from a number of programmers and some language characteristics. For example, we consider it to be easier to implement security measures with references rather than pointers. Next, the garbage collector reduces the risk of memory leaks and segmentation faults.

On the other hand, references and garbage collector lead to increased memory demands and processor time spent in finalizing and freeing unreferenced objects. In addition, memory fragmentation boosts incurred speed penalty. Therefore, the server must be written in a C-like language, as well as frequently executed active program code. GCC-like optimized code in processor-native instruction set is essential.

Java was an adequate choice for creating the proof of the concept for particular aspects of active networking. However, the situation has changed. To prove the concept of active networks as feasible for a productive use, we have to come close to performance of today IP stack implementations.

We succeeded with implementation of key components, while addressing shortcomings of preceding active-network implementations. Now, we need to finish the C++ port and to subsequently improve the code-execution speed with byte-code transformation.

Acknowledgment

The following members of SAN team implemented AntNet, Rendez-Vous, security and Windows IP Tunneling features: Zdenek Vacek, Miroslav Hendrych, Vladimir Aubrecht, Vaclav Papez and Petr Jaros {zdvacek|picard|aubrechv|vpapez|jarosp@students.zcu.cz}.

References

[1] J. Sykora and T. Koutny, "Enhancing Performance of Networking Applications by IP Tunneling through Active Networks", Proceedings of the Ninth International Conference on Networks, Les Menuires, France, 2010

[2] K. Calvert, "Reflections on Network Architecture: an Active Networking Perspective", In ACM SIGCOMM Computer Communication Review, Volume 36, 2006, pp. 27-30, doi: 10.1145/1129582.1129590

[3] D. L. Tennehouse and D. J. Wetherall, "Towards an Active Network Architecture", Proceeedings of DARPA Active Networks Conference and Exposition (DANCE.02), San Francisko, California, USA, 2002

[4] M. Maimour and C. D. Pham, "AMCA: An Active-based Multicast Congestion Avoidance Algorithm," Proceeedings of Eighth IEEE Symposium on Computers and Communications, Antalya, Turkey, 2003

[5] V. Ferreria, A. Rudenko, K. Eustice, R. Guy, V. Ramakrishna and P. Reiher, "PANDA: Middleware to Provide the Benefits of Active Networks to Legacy Applications, Proceedings of DARPA Active Networks Conference and Exposition, San Francisco, California, USA, 2002

[6] D. J. Wetherall, J. Guttag and D. Tennenhouse ANTS: A Toolkit for Building and Dynamically Deploying Network Protocols, IEEE Open Architectures and Network Programming 1998, San Francisco, California, USA, 1998

[7] T. Koutny et al., "Smart Active Node", http://www.san.zcu.cz/ Last Accessed on January 12, 2011

[8] S. F. Bush and A. B. Kulkarni, "Active Networks and Active Network Management – A Proactive Management Framework", Kluwer Academic/Plenum Publishers, 2001

[9] R. Rusty and W. Harald, "Linux Netfilter Hacking HOWTO", 2002, http://www.netfilter.org/documentation/HOWTO//netfilter-hacking-HOWTO.html Last Accessed on January 12, 2011

[10] G. di Caro and M. Dorigo, "An Adaptive Multi-Agent Routing Algorithm Inspired by Ants Behavior", Proceedings of PART98 - Fifth Annual Australasian Conference on Parallel and Real-Time Systems, Adelaide, Australia, 1998

[11] S. Chandra, U. Shrivastava, R. Vaish, S. Dixit, M. Rana, "Improved-AntNet: ACO Routing Algorithm in Practice" , Proceedings of UKSim 2009: 11th International Conference on Computer Modelling and Simulation, Cambridge, England, 2009

[12] L. Zhang and L. Xiaoping, “The Research and Improvement of AntNet Algorithm”, Proceedings of 2nd International Asia Conference on Informatics in Control, Automation and Robotics, Wuhan, China, 2010

[13] M. Hicks, J.T. Moore, D.S. Alexander, C.A. Gunter and S.M. Nettles, “PLANet: an active internetwork”, Proceedings of Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies, New York, NY, USA, 1999

[14] P. Menage, “RCANE: A Resource Controlled Framework for Active Network Services”, Proceedings of the First International Working Conference on Active Networks, Berlin, Germany, 1999

[15] D.S. Alexander, P.B. Menage, A.D. Keromytis, W.A. Arbaugh, K.G. Anagnostakis and J.M. Smith, “The Price of Safety in an Active Network”, Journal of Communications and Networks, Special Issue on Programmable Switches and Routers, Volume 3, Number. 1, March 2001

[16] W. Eaves, L. Cheng, A. Galis, T. Becker, T. Suzuki, S. Denazis, C. Kitahara, “SNAP Based Resource Control for Active Networks”, Proceedings of IEEE Global Telecommunications Conference, Taipei, Taiwan, 2002

[17] J. Gray, “Google Chrome: The Making of a Cross-Platform Browser”, In Linux Journal, Volume 2009, 2009

[18] Ch. Reis, A. Barth and Ch. Pizano, “Browser Security: Lessons from Google Chrome”, In Communications of the ACM, Volume 52, 2009, pp. 45-49, doi: 10.1145/1536616.1536634

[19] B. Yee, D. Sehr, G. Dardyk, J. B. Chen, R. Muth, T. Ormandy, S. Okasaka, N. Narula, and N. Fullagar, “Native Client: A Sandbox for Portable, Untrusted x86 Native Code”, Proceedings of 2009 IEEE Symposium on Security and Privacy, Oakland, California, USA, 2009

[20] M. S. Hossain, and A. El Saddik, “QoS Requirement in the Multimedia Transcoding Service Selection Process”, IEEE Transactions on Instrumentation And Measurement, Volume 59, Number 6, June 2010

[21] C. Xiao-lin, Z. Jing-yang, D. Han, L. Sang-lu and C. Gui-hai, “A Cluster-Based Secure Active Network Environment”, In Wuhan University Journal of Natural Sciences, Volume 10, Number 1, 2005, pp. 142 – 146, doi: 10.1007/BF02828636

[22] C. Percival, “Cache Missing for Fun and Profit”, Proceedings of BSDCan 2005, Ottawa, Canada, 2005

[23] H. Oi, “A Comparative Study of JVM Implementations with SPECjvm2008”, Proceedings of 2010 Second International Conference on Computer Engineering and Applications (ICCEA), Bali Island, Indonesia, 2010

[24] Sun Microsystems, Inc., “Java SE 6 Performance White Paper”, http://java.sun.com/performance/reference/whitepapers/6_performance.html#2 Last Accessed on January 12, 2011

[25] T. Koutny and J. Safarik, "Load Redistribution in Heterogeneous Systems", Proceedings of the Third International Conference on Autonomic and Autonomous Systems, Athens, Greece, 2007