# WinJTAP Interface for Multicast Middleware on the Win32 Platform

Computer Science Project

Submitted by:
Milan Nikolic
2008

Head:
Prof. Dr. Torsten Braun

Assisted by:
Dragan Milic
Marc Brogle

Computer Networks and Distributed Systems Research Group
Institute of Computer Science and Applied Mathematics (IAM)
University of Bern

## Abstract

This project report describes the implementation and performance evaluation of an interface, called WinJTAP, which allows Java applications to use Ethernet frames captured by a virtual network interface, called TAP, on the Microsoft Windows32 (Win32) Operating System. Since Win32 and TAP are written explicitly in the C/C++ programming language, we require an additional environment, which enables the communication of C/C++ with the Java Virtual Machine (JVM). Those two environments can be coupled using the Java Native Interface (JNI), a special kind of an interface, which allows the JVM to use libraries, functions and routines written in the C/C++ programming language. WinJTAP will be used for the evaluation of solutions for several known network problems, such as fast IP traffic dissemination or Application Layer Multicast (ALM) on the Win32 platform. In particular, WinJTAP is developed for use with the Multicast Middleware application, a Java based P2P application, whose main purpose is to bridge ALM and IP Multicast through encapsulation of IP packets (multicast traffic) into Ethernet 802.3 frames. Implementing WinJTAP for Win32 is rather difficult, due to the complexity of the network communication architecture of Win32 and lack of source availability. A similar implementation of this kind of interface already exists for UNIX based operating systems such as Linux and MacOSX. The evaluation of performance and stability of WinJTAP is accomplished using various software analysis tools, such as JTAPHub (software emulation of network hub) and Multi-Generator (MGEN). Maximum measured data throughput using WinJTAP on Win32 was 19,6 MBit/s, which shows that WinJTAP can be used by the Multicast Middleware concerning performance.

# Table of Contents

# 1. Introduction and Motivation

With growing communication bandwidth requirements, the way of communication in the Internet has rapidly changed. Internet Service Providers are providing nowadays Digital Subscriber Lines (DSL) with high bandwidth. The exchange of multimedia content, such as audio and video streams, became typical in the world of Internet communication.
The fact that the Internet is a packet switching network and that unicast connections between *n* peers can cause redundancy, lead to the consideration of using IP Multicast as a more efficient way to transfer data packets. The main principle of this new communication paradigm is replication of data packets on routers, where they can be sent to many recipients.
As the deployment of IP Multicast never found broad application in practise, the idea of emulating IP Multicast in user space has been raised. As a result, Application Layer Multicast (ALM) has been developed.
The main advantage of ALM over IP Multicast is the independency from the underlying network (multicast functionalities are implemented explicitly on end-systems).
This approach does not require any modifications of the network or router architecture.
ALM has also some disadvantages, such as data packet replication on end-systems (due to the use of unicast connections) and performance, and therefore is less efficient than IP Multicast. Research at the University of Bern has shown that combining IP Multicast with ALM would bring the best performance-efficiency results, considering the difficulty of any modification of network infrastructure or OS kernel. A typical example of using IP Multicast in combination with ALM is the Multicast Middleware (MM), described in [1, 6]. MM is a Java application, which enables IP Multicast traffic on end-systems, using only unicast connections, and is a part of the EuQoS project [2], whose main aim is to support end-to-end QoS for heterogeneous networks. MM uses ALM for transportation of multicast traffic and a software emulated network device, called TAP [5, 14] for providing IP Multicast to applications. This virtual network interface establishes a direct communication link between the physical and application layer by capturing of Ethernet 802.3 frames from the physical layer. Using TAP, MM does not require any modification of the OS kernel, such as modifying the Network Driver Interface Specification (NDIS) [8, 9] architecture on Win32 and makes the processing of multicast traffic transparent to all applications.
The motivation for this project was to develop and evaluate a new interface, which would establish the communication between TAP and the MM on the Win32 OS. This form of communication can be achieved using interception of captured Ethernet frames and forwarding the intercepted Ethernet frames to the user space, where they can deliver the encapsulated Multicast traffic, which then can be processed by an application. With the described principles, an efficient IP Multicast traffic dissemination can be achieved. The implementation of this kind of interface, called WinJTAP, requires also a special kind of interface, called Java Native Interface (JNI) [10], which offers the possibility to integrate legacy code written in the C/C++ programming language with Java applications. Using JNI is necessary, due to the Java base of the MM and the C implementation of TAP. Important issues that have to be considered are stability and performance. For the evaluation of the basic functionality of WinJTAP, we needed a test application, which would be used to simulate the communication scenario between TAP and the MM.

As a result, we implemented a software emulation of a network hub, named JTAPHub, a Java application, which has the possibility to establish a “Full Mesh P2P group” and to achieve exchanging high bandwidth data using WinJTAP. For detailed evaluation of WinJTAP (performance and stability), we used a Video LAN Client (VLC) [12] and Multi Generator (MGEN) [7] in combination with JTAPHub.
This project report consists of four parts. In Section 2, technologies used in our implementation are presented: a short introduction to the TAP interface on the Win32 platform, as well as the architecture and the communication with the TAP interface. Further, the concept of I/O control in Win32, synchronization of I/O and JNI are described. Section 3 describes the implementation of the WinJTAP interface and its architecture. In Section 4 the architecture of the JTAPHub test application is described. Section 5 evaluates performance and stability of WinJTAP. Finally, conclusions and an outlook are presented in Section 6.

# 2. Technologies used for Implementing WinJTAP

## 2.1 TAP Interface

### 2.1.1 Overview

A TAP interface is a software emulated 802.3 network interface, seen by the operating system as a normal Ethernet device. However, instead of receiving packets from physical media, it receives packets from a user space program and instead of sending packets via a physical medium, it sends packets directly to a user space program (see Fig. 2.1). The user space program sends data in form of Ethernet frames directly to a hardware network device and data is being sent to the receiver over a physical link. On the receiver side, data is received from the network device and sent to a user space application that forwards the data directly to a TAP interface. With this kind of communication between an application and the network device, the data packets are captured directly after the application layer and sent to a physical layer.

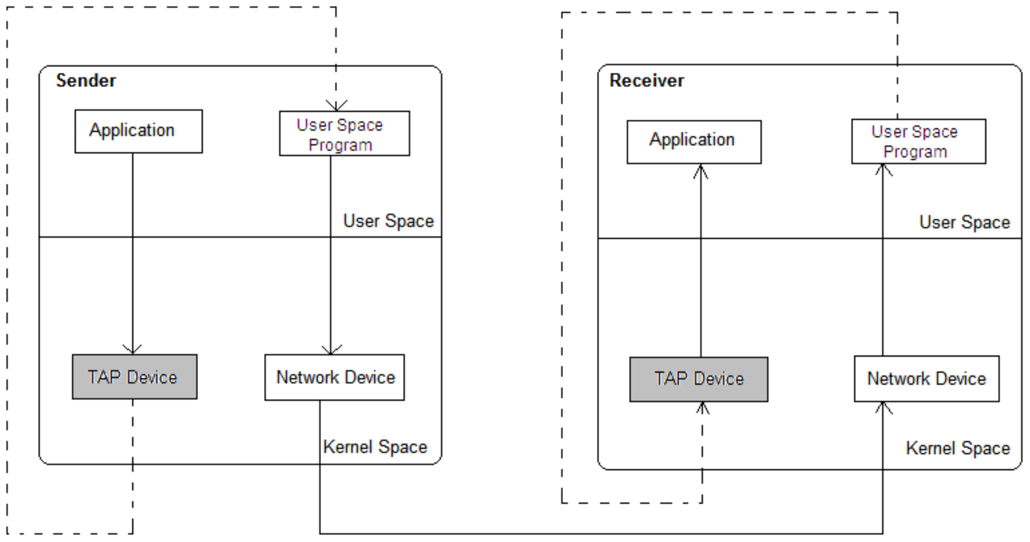


Figure 2.1 Sending and Receiving Data using the TAP Interface on Win32 OS.

### 2.1.2 TAP Adapter Parameters in Windows

The initiation of the communication with TAP interface on the Win32 OS can be established using "registry parameters". For the communication with any network device on Win32 OS, two parameters are needed*:*

1. *Adapter parameter*: Stores the information about a device
2. *Network Connection parameter:* Describes the connection made with a device

As for all other network devices on Win32, the TAP adapter information is stored in the Registry Database, which is used for storing important information (e.g. variables) used during run-time.

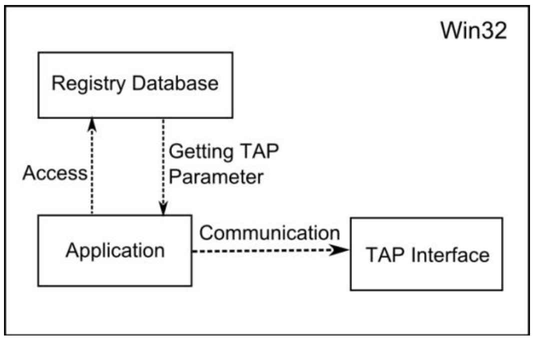


Figure 2.2 Accessing the Registry Database for the necessary Parameters used for the Communication with TAP Interface

### 2.1.3 **Architecture of TAP**

The TAP interface has been mainly developed for TCP/IP tunnelling (capturing of encapsulated IP packets) and it is used in applications such as VTun and OpenVPN [14]. As mentioned in the Introduction, the main purpose of TAP for this project was also the encapsulation of IP Multicast traffic. For our project, we used OpenVPN's version of the TAP interface. The TAP-Win32 driver is distributed with OpenVPN 1.5-beta5 and later, which is derived from Cipe-Win32 [13], a software for crypto IP-encapsulation. For a detailed overview about the TAP interface's architecture in OpenVPN see the Class Diagram presented in the UML language (Appendix, Fig. A.3.1).
As shown in section 2.1.1, the communication with TAP differs from the communication with hardware network devices, thus the difference is not visible on the application level. A hardware network device sends data starting from the physical layer, and this data goes through all network layers in the OS to the application. On the other hand, the TAP interface achieves direct communication between application layer (OSI layer 7) and physical layer (OSI layer 2) through capturing Ethernet frames, and sending them to the user space, where they can be directly processed by a user space application (see Fig 2.2).

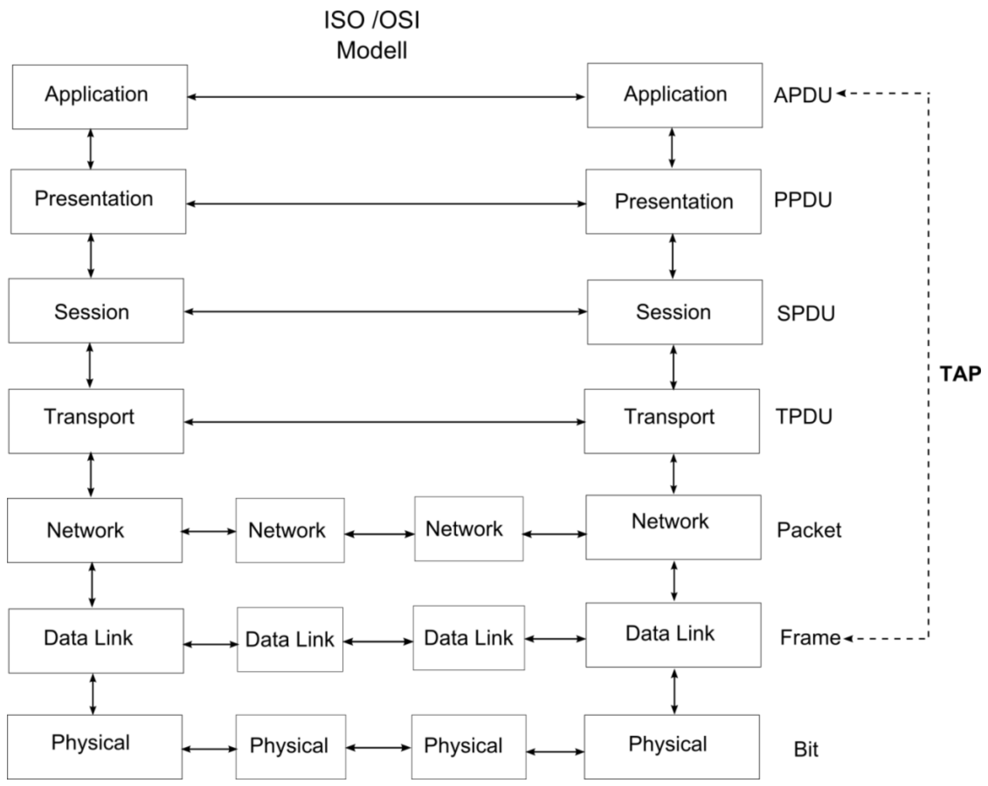


Figure 2.2 Transporting Data with a TAP Interface in the OSI/ISO Model

### 2.1.4 **Network Driver Interface Specification (NDIS) in Win32**

Since we introduced the communication of TAP interface on the application level, we also need to introduce the other side of the communication path, namely the communication with the hardware device. For understanding purpose, we will discuss the main principles of the communication with a Network Interface Card (NIC) in Win32.
We will introduce the Network Driver Interface Specification (NDIS) [8, 9], which plays an important role for the communication of Win32 with NIC.
NDIS is a set of communication rules (protocols), which have to be applied during the communication of higher level drivers on the Transport Driver Interface (TDI) layer, such as transport protocol drivers (e.g. TCP/IP) with network device drivers, which are establishing a direct communication with a network device. With this kind of communication (with a modular architecture), high-level drivers are independent from NIC.
NDIS consists of the following components:

1. NDIS Miniport driver
2. NDIS Library
3. NDIS Protocol drivers

A NDIS miniport driver has the task to manage the communication with a network device and to establish the connection with higher level drivers, such as intermediate drivers and

transport protocol drivers and it is fully independent from the OS. NDIS miniport driver is dedicated to a device, and has also the task to manage “plug and play” functionalities for network devices.
The communication between the miniport driver and high-level drivers (NDIS protocol drivers) is achieved through the NDIS library or NDIS “Interface Wrapper”, which is OS specific. The NDIS Wrapper consists of OS specific functions (called *MiniportXxx* functions) that encapsulate the OS functions needed for the communication with the miniport driver (see Fig. 2.3).

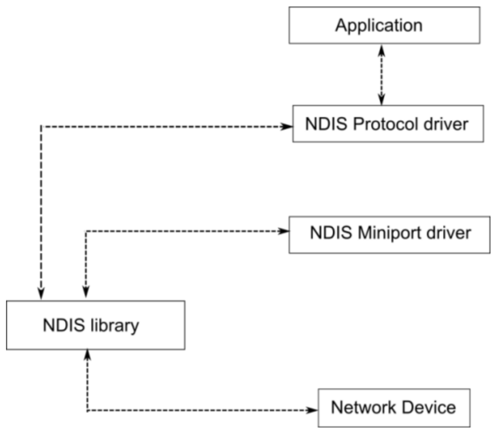


Figure 2.3 Communication of an Application with a Network Device on Win32 using the NDIS Library, NDIS Miniport Driver and High Level Drivers

The NDIS protocol driver is used for binding the upper layers, such as the layer (TDI) with lower edge layers (NDIS Wrapper interface). NDIS protocol drivers use NDIS specific functions (so called *ProtocolXxx* functions) for communication with the NDIS wrapper. This communication is achieved using a “virtual” miniport driver (intermediate driver) which is exported by the NDIS wrapper and bound with the protocol driver.

As seen in the former description, the communication of Win32 with a network device is achieved using NDIS. Since the TAP interface should intercept Ethernet packets, it will also use NDIS for the communication with a network device. TAP interface will use the miniport driver and NDIS wrapper (intermediate driver), for the communication with a network device (see Fig. 2.4). Protocol drivers cannot be used for this kind of communication, since they are managing the communication with upper layers.

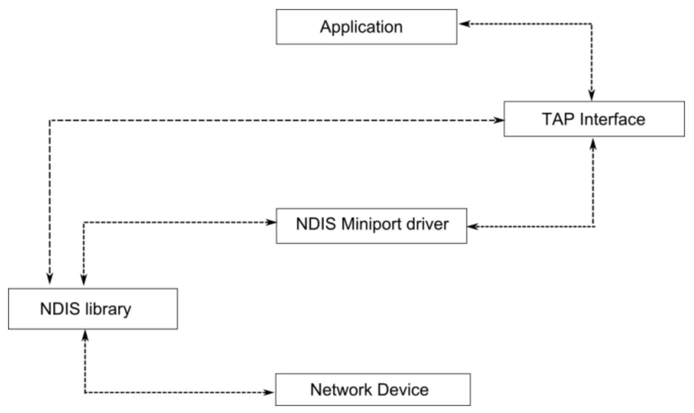


Figure 2.4 Communication of an Application with the Network Device on Win32 using NDIS Miniport Driver and TAP Interface

## 2.2 Device Input and Output Control

In operating systems such as UNIX or MS Windows, input and output devices are controlled by “Device Input and Output Control” (IOCTL), a control interface, which manages the communication between hardware devices and applications. The communication is achieved by sending the control parameters (codes) from user space to a specific device driver in kernel space and further to a hardware device. A typical example of using IOCTL in Win32 is transferring data from a Floppy Disk Drive (FDD) to the Hard Disk Drive (HDD). In order to start transferring data from FDD to HDD, IOCTL has to be used for the initialization of the I/O processes (assigning the control code to the device). After the initialization, the OS can start with calling other I/O functions, such as *CreateFile*, *ReadFile* or *WriteFile* for transferring data. The *CreateFile* function opens the device (e.g. FDD) using the *hDevice* handle, which is delivered from the IOCTL function.
In Win32, a device I/O control function is called *DeviceIoControl* (see also Fig. A.2.5, Appendix).

It takes 8 parameters:

1. *hDevice*: A Handle to the Device, that is to be controlled (used by *CreateFile* function)
2. *dwIoControlCode*: I/O control code that determines which operation is to be performed
3. *lpInBuffer*: Pointer to an input buffer that contains data for the I/O operation
4. *nInBufferSize*: Size of the input buffer
5. *lpOutBuffer*: Pointer to an output buffer that contains the data returned from the I/O operation
6. *nOutBufferSize*: Size of the output buffer
7. *lpBytesReturned*: Pointer to a buffer, which contains the size of received data from the I/O operation
8. *lpOverlapped*: Flag, which indicates if the Overlapped Structure is to be used (see Section 2.3.2 for more details)

## 2.3 I/O Synchronization

### 2.3.1 **Overview**

In this section we will describe the concept of I/O synchronization, an important mechanism for achieving certain performance of data transfer with WinJTAP. Since we are using TAP interface and Network Interface Card as I/O devices for transferring the data, we have to consider this principle in order to avoid unnecessary waiting time and eliminate possible performance bottlenecks. In general, synchronization in I/O is needed in the case of accessing the same resource from many (I/O) processes at the same time. This situation occurs often in multitasking systems with multiple threads active in the same time interval. I/O synchronization assures that there is some order during the access (of threads) to the resources during the I/O process and avoids possible deadlock situations. Operating Systems usually use “Locks” or “Semaphores” (described in Section 2.3.4) for I/O synchronization. For example, if a thread A accesses a variable X for writing, and a thread B wishes also to access the same variable, a lock on variable X should give the signal that this variable is busy and therefore not accessible. Thread B will have to go into a “wait-state” until it receives the signal that variable X is free for accessing (writing).

### 2.3.2 **I/O Synchronization Types**

In order to understand I/O synchronization principles and types, we explain shortly the flows occurring during I/O processing in Win32 with a simple example.
Let us assume that one application needs to read data from an optical drive (e.g. CD-ROM). In the first step, the initialization of the I/O process with IOCTL occurs (a system call is sent to the kernel space and then to the I/O device). The I/O device responds, and data transfer from the device to the application can start. Data, which is transferred from the kernel to the user space, has to be processed (serialized) in order to be usable for the application. Here we can notice two independent occurrences during I/O processing:

1. I/O operations (e.g. sending a system call or transferring data from a device to the user space)
2. Processing of data in the user space

In the multitasking systems, where an application can send more than one I/O request to a device (multiple threads), the situation can be more complex. Here we can use different approaches. With the first approach, the active thread will block an application during the time of I/O processing. Other threads will have to wait for the termination of the I/O operations. This approach is not performance optimized, since other threads have to wait for termination of the I/O processing. Another approach is based on the fact that the CPU speed is normally faster than the speed of the I/O device. Considering this fact, two main processes (I/O operations and data processing) can be performed quasi-parallel, without blocking an application. After sending the I/O request to a device, a thread goes back immediately to process the data in the user space, since this can be done very quickly. The only part that would be needed for this mechanism is some kind of additional structure,

which would enable the informing about completion of the parallel data processing (checking the current state of the objects). Of course this data structure exists in Win32 and is called “Overlapped Structure”. Overlapped Structure is described in the section 2.3.3. Using this approach, an application can allow processing of data during its own I/O operations, which is performed in the background.
Depending of the choice of the described approaches, we distinguish two types of I/O synchronization:

1. Synchronous I/O
2. Asynchronous or overlapped I/O

For the synchronous I/O, a thread starts an I/O operation and immediately enters a wait state until the I/O request has completed. It is not possible to achieve overlapping for data processing and I/O operations at the same time (see Fig. 2.3).

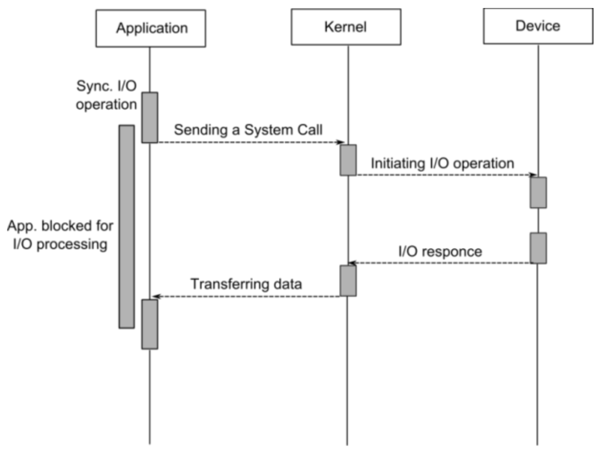


Figure 2.3 Communication between an Application (User Space) and a Device through the Kernel Space using Synchronous I/O

For asynchronous I/O the thread continues processing another job until the kernel signals the completion of the I/O operation. With this principle, asynchronous I/O permits over-lapping of processing of data and I/O operations (see Fig. 2.4).

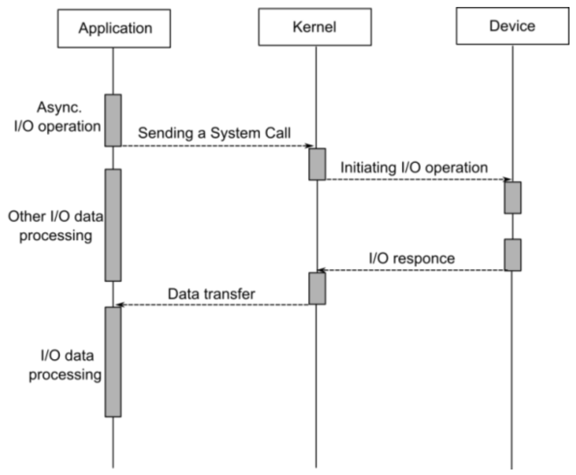


Figure 2.4 Communication between an Application (User Space) and a Device through the Kernel Space using Asynchronous (Overlapped) I/O

### 2.3.3 **Overlapped Structure**

The Overlapped Structure is a Win32 API data structure, which is used in asynchronous I/O functions where several simultaneous overlapped operations, such as *ReadFile* or *WriteFile* are performed (see Fig A.2.1 in the Appendix). Every time when (asynchronous) reading/writing processing occurs, the Overlapped structure is passed to the corresponding I/O functions. The main purpose of this structure is to deliver information (state) about I/O completion. The I/O functions can retrieve the information about I/O process completion by specifying the *hEvent* flag, which is used for the current state of the signalling objects. If the I/O function is pending for some reason, a *WaitForSingleObject* function is used. This function has to wait for completion until the object state has been changed (signalled) or some time interval has been reached. If some other I/O request occurs during the pending time of the *WaitForSingleObject* function, the I/O function cannot be called and it returns FALSE state. Thus the same function can also deliver FALSE state due to an I/O error. To be able to distinguish an I/O error from a pending state, a special error code (ERROR_IO_PENDING) is used (see Fig. 2.5). If the return value of this error code is TRUE, it means that the I/O function is pending and *WaitForSingleObject* function will be called in order to put the I/O process into a wait state. When the waiting time of the I/O process elapses, the function *GetOverlappedResult* retrieves the result of the overlapped I/O function (see Fig. 2.5).

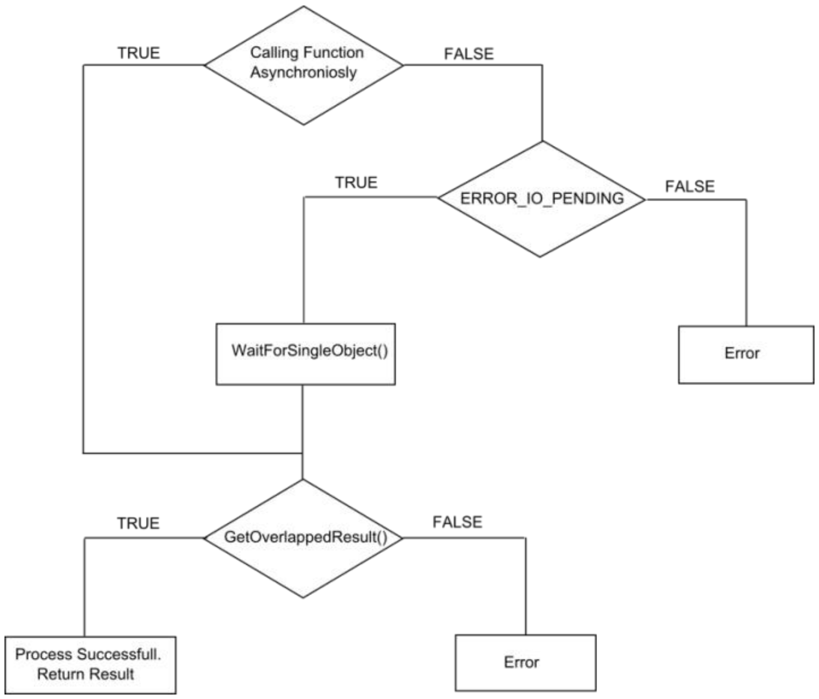


Figure 2.5 Handling Overlapped Structures during the Asynchronous I/O Operation in Win32

## 2.4 Java Native Interface

### 2.4.1 Overview

Platform independence demands its tribute with Java. In practice, developers always push the borders of Java. It often happens that the specific possibilities of an operating system cannot be used. On the other side, native applications would also wish sometimes to be able to use the advantages of Java. Java Native Interface (JNI) makes this possible. With mutual access to libraries, which are provided in Java and C/C++, JNI can unite those two programming languages.
The main advantage of JNI is the possibility to allow Java programs to use the libraries, functions and routines written in C/ C++. The reverse way is also possible; Java Runtime Environment (JRE) can be called by JNI from C/C++ programs (see Fig. 2.6).
With JNI we can even catch and pass the exceptions in native methods to Java. Both environments, native (C/C++) and Java can access Java objects, they can produce new Java objects, change their method calls and even exchange Java objects mutually.

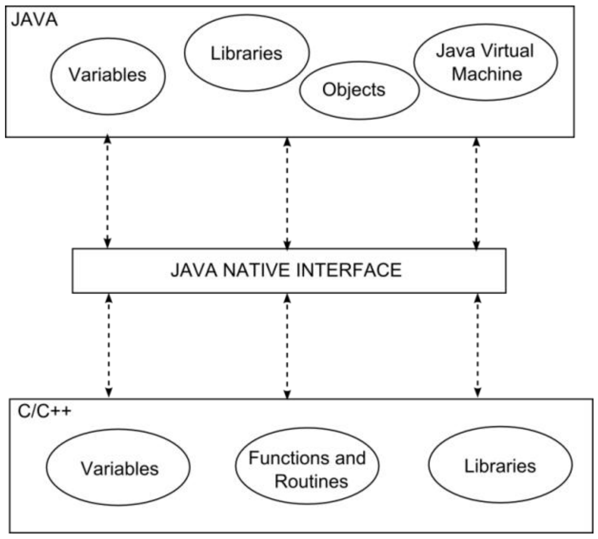


Figure 2.6 JNI as coupling Interface between Java and C/C++

### 2.4.2 Application of JNI

JNI is used in situations, where parts of the functionality are computer-bound and therefore written in C/C ++ or assembler. In practice, the call of C libraries from a Java program happens more frequently than the call of Java from C. This direction of the communication is exactly what we used in the WinJTAP implementation. Therefore, this call structure will be the center of our attention.

### 2.4.3 JNI Creation Process

The creation process of JNI can be described as following:

a) Creating a Java Class (declaring native methods).
b) Compiling the Java source file.
c) Generating the C-compatible Native Method Header file from the Java code with the “javah” tool.
d) Writing the C/C++ implementation of the native method.
e) Compiling source and header files to a dynamic library.
f) Starting the Java class and calls of the native methods.

For a detailed overview about the creation process, see also the diagram in the Fig. A.2.1 .

### 2.4.4 Declaring the Native Methods

A native method declaration contains the *native* modifier, which means that a specific method is implemented in another programming language. This statement is comparable with one “external” declaration of an external function under C/C++. For each function, which has to be addressed in an external library, a native declaration has to be declared in the Java program. There is no implementation of the native method in the Java Class itself (see Fig. 2.7).

```
class HelloJNI
{
        private native void print();
        public static void main(String[] args)
       {
              new HelloJNI().print();
        }
        static
       {
              System.loadLibrary("HelloJNI");
        }
}
```
Figure 2.7 Example of Declaring Native Methods in Java

In Fig. 2.7, we can also notice the *System.loadLibrary* function call, which binds dynamically the external libraries during run-time with a name as an argument. Those libraries are so called “shared libraries”, which are generated when compiling the C code. The actual library, which has to be found, is platform dependent. In Win32, this library is called Dynamic Link Library (DLL). The example in Fig. 2.7 shows how to load the library “HelloJNI.dll” in the Win32.

### 2.4.5 **Creating the Native Method Header File**

The new Java class file *HelloJNI.class*, generated by compiling the Java code serves now as starting point for the generation of the C-header file.
In order to generate a JNI header file, we have to use the *javah* tool, provided in the Java Development Kit (JDK) (see Fig. 2.8). The JNI header file contains the prototypes of the C/C++ functions (see Fig. 2.9) and therefore supports writing the C/C++ implementations for the native methods.

```
javah -jni HelloJNI
```
Figure 2.8 Generating the Native Method Header File with the *javah* Tool

```
JNIEXPORT void JNICALL
Java_HelloJNI_print (JNIEnv *, jobject);
```
Figure 2.9 Generated Header File Methods

### 2.4.6 Native Method Implementation

Implementation of the C/C++ functions can be done using the prototypes from the generated header file (see Fig. 2.9). Observing the example in Fig. 2.9, we notice two important lines.
The first line contains two macros: JNIEXPORT and JNICALL, defined in the *jni.h* header file, which are used to notify a C-compiler that the function should be exported from the native library and the appropriate code should be generated in order to call this function properly. The second line contains the name of the C function, which is formed by concatenating the “Java_” prefix with the class name and the method name. In our example, this function will look as follows: “Java_HelloJNI_print”.
The body of the function has a default implementation in C.
Of course the implementation should be accommodated according to the given native method arguments in the function. In our example we do not have any additional native arguments, except the default native arguments (see Fig. 2.10).

```
#include <jni.h>
#include <stdio.h>
#include "Helloworld.h"

JNIEXPORT void JNICALL
Java _HelloJNI_print(JNIEnv *env, jobject obj)
{
     printf("Hello JNI!\n");
     return;
}
```

Figure 2.10 Implementation of the Function in C

### 2.4.7 Native Method Arguments

As seen in the Section 2.4.6 (Fig. 2.10), native methods have two standard parameters, which cannot be seen in the java native method declarations. The first parameter is “JNIEnv *” that points to the JNI Environment-Interface, a pointer, which actually points to another pointer that points to the array of JNI functions or function table (see Fig. 2.11). The second parameter is “jobject”, which points to the Java Object on which this JNI function is invoked. The “jobject” parameter corresponds to the “this” pointer in C++. In our example the Java Object would be the “HelloJNI” Java Class Object.

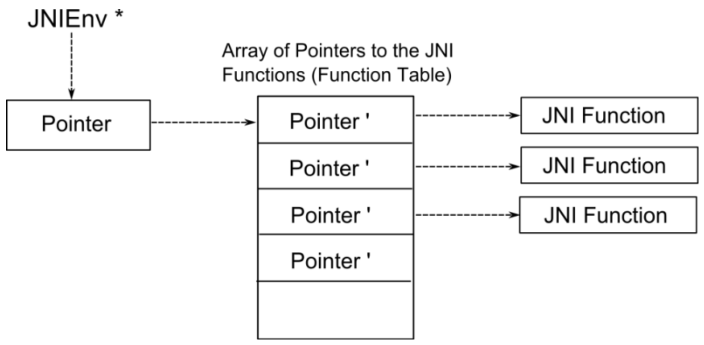


Figure 2.11 Role of the JNIEnv * Parameter during the Communication of the C Functions with Java Virtual Machine

Other native method arguments, which are often used in JNI functions, are: “jint”, “jstring” and others.

### 2.4.8 Creating the Native Library

After the implementation of the functions in C/C++, the native library (described in Section 2.4.4) has to be generated in order to load C/C++ functions (see Fig. 2.11).

```
cl -IC:\Program Files\java\include -IC:\Program Files\java\include\win32
-MD -LD HelloJNI.c -Fe HelloJNI.dll
```

Figure 2.11 Building a Dynamic Link Library (DLL) using C/C++ Compiler in Win32

For more detailed information about JNI refer to [10].

# 3. WinJTAP Implementation

## 3.1 Overview

Implementing the WinJTAP interface requires understanding the TAP architecture and communication with applications (e.g. OpenVPN) in Win32, and also the concepts of concurrent reading/writing and I/O synchronization.
The required concepts (described in Sections 2.1, 2.2 and 2.3) are already available in Win32 and can be directly called using the C environment.
For performance reasons, such as fast IP packet dissemination, asynchronous I/O is used.
With the help of JNI, described in the Section 2.4 we are able to call TAP related functions (e.g. reading from a TAP interface) directly from the Java application and to intercept the raw data, read from a TAP interface.

## 3.2 Architecture and Design of WinJTAP

The architecture of WinJTAP is divided into two main parts. The first part is a C environment (low level communication part), which establishes the communication base with a TAP interface (e.g. opening TAP, reading from TAP). The second part would be the JNI part, which establishes the communication between C and Java environments and redirects the data (buffer content) delivered by the C environment from the TAP communication.
The communication of a Java application with TAP through WinJTAP is initiated by the Java application. The Java application triggers the native functions, implemented in the C environment (using JNI) and establishes the communication with TAP. The concepts, described in Sections 2.1, 2.2 and 2.3 are used for the C environment, since they can be applied only at this communication level, due to strong coupling with the Operating System. (See Fig. 3.1). At the higher level of the communication with TAP, JNI as described in Section 2.4 is used.

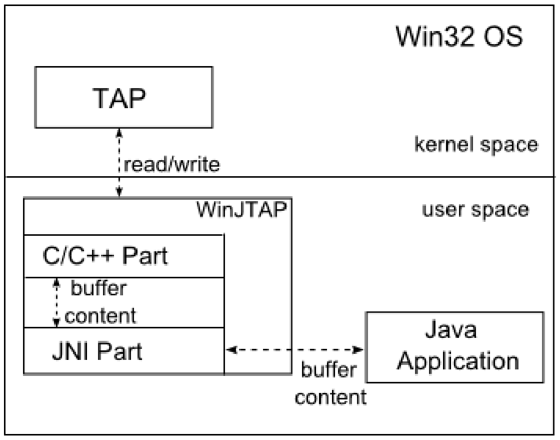


Fig 3.1 WinJTAP's Role during the Communication Establishment between the Java Application and TAP in Win32

### 3.2.1 **Communication of WinJTAP (user space) with TAP (kernel space)**

In this subsection we describe the communication process between TAP (kernel space) and WinJTAP (user space). At the first stage of the communication process, WinJTAP locates the driver of the TAP interface in the Registry Database and collects all necessary information for the communication like Device ID or MTU. After the information was successfully collected, the “opening process” for the TAP interface can start. As every I/O device, a TAP interface can be opened in Win32 with the system call *CreateFile*. Before calling the *CreateFile* function, the I/O process has to be initialized. The initialization of I/O processes in Win32 is accomplished using the *DeviceIoControl* function (see Section 2.2). Since we require the Asynchronous (Overlapped) I/O for reading and writing from/to the TAP device, we need to set the appropriate flag for Overlapped I/O (*lpOverlapped)* in the *DeviceIoControl* function and also to pass the “Overlapped Structure” to the *CreateFile* function. After the successful opening process of the TAP interface, the reading/writing process can start.

### 3.2.2 **WinJTAP`s C/C++ Part**

This subsection describes the mechanisms, which control the I/O functions activated with starting the communication with TAP (described in Section 3.2.1). The most important issue was avoiding unnecessary waiting time caused by a concurrency, which occurs at the reading and writing processes. Avoiding unnecessary waiting for I/O processes assures the best throughput performance. In WinJTAP we used a queue for temporarily storing the data read from TAP, in order to be able to switch promptly to the pending writing process. The temporarily stored data can be consumed by an application in the user space. With this principle, we are able to achieve overlapping in the reading and writing processes. This affects the performance of WinJTAP.
The standard queue operations, such as accessing the buffer, reading from buffer, clearing the buffer and putting the empty buffer on the beginning of the queue require also synchronization. For this reason Critical Sections and Semaphores have been used. Semaphores are used for signalization and Critical Sections for protecting the multiple threads from accessing the same buffer. The principle of using Critical Sections and Semaphores is described as follows:
If the buffer content (queue unit) has been accessed by a thread, it has to be declared as a critical section, in order to signal the other threads that access is not allowed. The other threads will go into a “Wait-State”, until they are notified (signalled) with a semaphore that the buffer is free for accessing. Once the thread has finished the I/O operations on the buffer, the critical section can be leaved.
The data (buffer content) taken from the queue, is then transferred to the application using JNI (see Fig. 3.2). The synchronization functions, such as Semaphores and Critical Sections are already implemented in Win32 and they can be used directly in the C environment. For more information about Critical Sections and Semaphores in Win32, refer to [8].

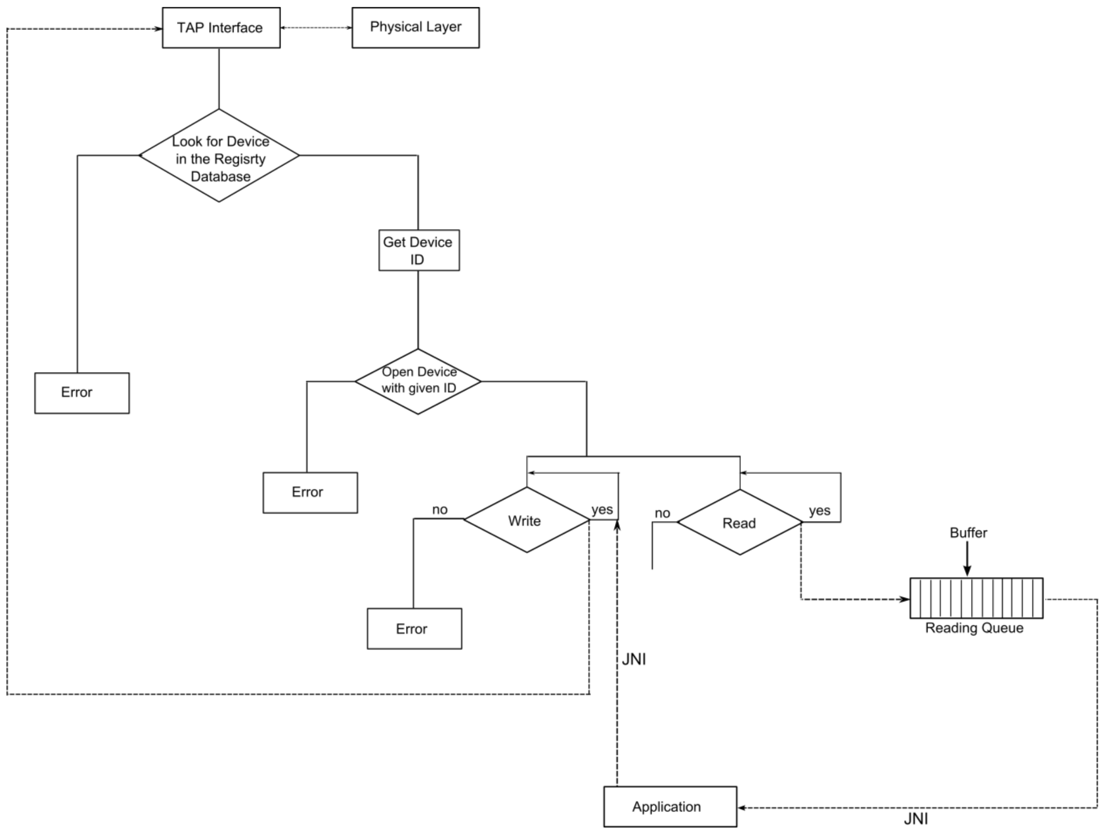


Figure 3.2 Architecture of WinJTAP using the standard communication base with TAP Interface in Win32.

### 3.2.3 **WinJTAP's JNI Part**

Once the communication with a TAP interface is established and the transfer of the captured data to the user space is done, we are able to start with further interception (redirection) of data to the Java application on a higher communication level with TAP. This redirection is done using JNI. JNI used so called native functions (methods) to call the I/O functions implemented in C environment. With this kind of communication the intercepted data can be transferred from one C environment to the Java environment.
JNI functions allow us also to control TAP interface from the Java side (e.g. open/close TAP).

## 3.3 Native Method Implementation

Implementation of JNI functions is accomplished using the concepts, introduced in Section 2.5. In general we implemented four basic JNI functions:

1. Read from TAP interface
2. Write to the TAP interface
3. Open the TAP interface
4. Close the TAP interface

See Fig. also A.4.1, A.4.2, A.4.3, A.4.4 and A4.5 in the Appendix.

# 4. JTAPHub P2P Ethernet Application

## 4.1 Overview

JTAPHub is a Java application, which simulates a network hub. It offers the possibility to connect peers into a “Full Mesh” P2P group where every peer can establish unicast (P2P) connection to the other peers in the group and can exchange (high bandwidth) data between group members. The main purpose of JTAPHub is to test the stability, robustness and basic functionalities of the WinJTAP interface. At the beginning, the focus has not been on performance.

## 4.2 JTAPHub Architecture and Design

As described, JTAPHub connects different peers using P2P connections. For establishing the P2P connections we used Java Sockets, which were used for establishing TCP connection between peers using the combination of one arbitrary port number and IP address. Since Java has the possibility to represent IP addresses as Java Objects (“InetAddress”) with different kind of parameters as arguments (e.g. port number), this can be directly applied in our application, together with Socket communication. Java Socket communication interfaces can be called directly from the “java.net” library class (for more information about Sockets in Java, please refer to [4]). The communication scenario between peers and the JTAPHub application can be described as follows:

For establishing a P2P group with JTAPHub, we need to have some peer, which would initiate this P2P group. A group initiator, also called “Master Peer”, is responsible for managing connections of new peers. Every peer that wants to connect to the group has to know the IP address of the Master Peer. This address can be supplied over e-mail or another communication method, since there is no other way for other peers to get the IP address of the Master Peer at the beginning. Let us assume that two peers want to create a P2P group. Before they can create this group, they have to identify a “Master Peer”. The address of the “Master Peer” is then known to the other peer that wants to connect to it. After establishing the connection to the “Master Peer”, the new member receives promptly broadcasted messages in form of a list (see Figure 4.1). This list consists of IP addresses of peers, which are connected to this group (in our case, it will be only the IP address of the “Master Peer”). With help of this list, a new peer can start to establish a direct (P2P) connection with all peers, whose address has been found in the list. (In our example, that would be only “Master Peer”). If the IP address of “Master Peer” is the only one that is found in the List Message, it will not be taken into account for the connecting, since this connection already exists. After successfully establishing connections to other peers in the group, the new member starts to receive broadcasted Data Messages (if some of the peers in the group are broadcasting them). Receiving and sending List and Data Messages is achieved using threads, which are always ready for receiving/sending data. The IP addresses of successfully connected peers in the group are also registered in the “ConnectedPeersList”.

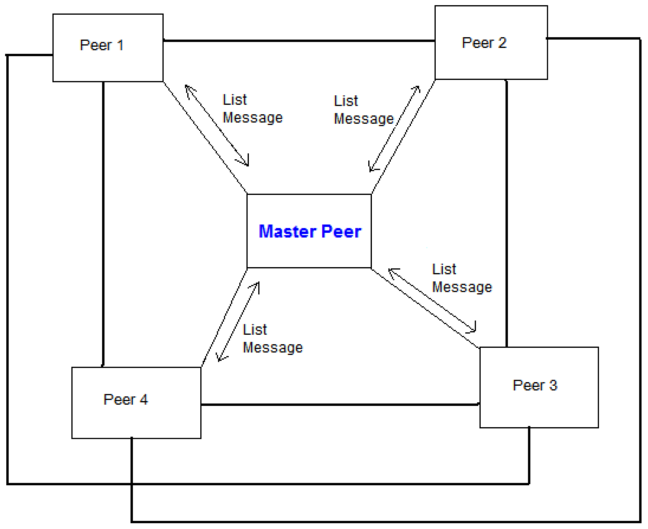


Figure 4.1 Peers Connected to the Master Peer Using JTAPHub and Exchanging List Messages

For sending the Data Messages, peers will have to look for the destination addresses in the “ConnectedPeersList. This list is constantly updated with List Messages, which contains updated information about all peers that are active in the group (also new ones). When one Peer receives a new List Message, it takes all IP Addresses from it and updates the “ConnectedPeersList” with the new IP addresses (see Figure 4.2).

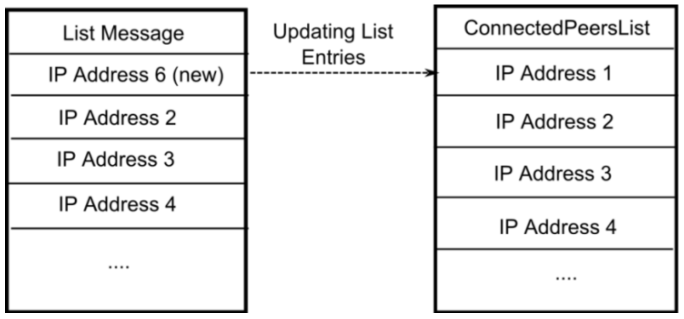


Figure 4.2 Updating the ConnectedPeersList with new entries from the List Message

Once that the “ConnectedPeersList” is updated, a peer checks for differences in the list (connections with new members of the group or cancelled connections from peers, that have left the group).
For example, if some peer finds out, that one “connected”  peer has left the group (IP address is deleted from ConnectedPeersList during the update), it closes automatically the socket connection to this peer and stops the sending and receiving threads. With this mechanism, we are avoiding unnecessary broadcasting of data to unreachable recipients. Leaving the group means, disabling the connection to the “Master Peer”. The mechanism, which disconnects automatically all connections to the other peer, is a thread, which periodically scans the List Messages and accord-

ing to the changes, disables the connections. It is assumed that the IP address of the Master Peer is always in the list. This is logical, because the Master Peer is the first peer that initiated the peer group. As seen, broadcasted data is also represented as messages, namely “Data Messages”.
Data Messages are generated Java objects with a serialization of the buffer content read from the TAP interface. On the other side, the buffer content can also be de-serialized from “Data Messages” for writing purposes. The buffer serialization is available in Java and can be easily applied to our application. For more details about Java buffer serialization, please refer to [4].
For the overlapping in I/O, we used queues (e.g. “LinkedBlockingQueue”), which are implemented in Java and can directly be used from the “java.util.concurrent” library (for detailed description about using queues in Java, refer to [4]). For a detailed architecture overview of JTAPHub, see the class diagram, presented in UML in Fig. A.6.1 in the Appendix.

# 5. Performance and Stability Evaluation

Since WinJTAP has been mainly realized to be used by the Multicast Middleware (MM) application, whose main purpose is high performance multicast dissemination, a certain performance has to be achieved in order to have this interface applicable for MM.
For evaluating WinJTAP, we used applications such as Video LAN Client (VLC), Multigenerator (MGEN) in combination with JTAPHub described in Chapter 4. JTAPHub was used for simulating a P2P group using WinJTAP, VLC for testing the basic functionalities and stability of WinJTAP and finally MGEN for performance evaluation, such as bandwidth throughput, dropped packets and jitter.

## 5.1 Video LAN Client (VLC)

### 5.1.1 Overview

Video Lan Client (VLC) is a free (under General Public Licence available) media player developed by the VideoLAN project [12]. It has the possibility to encode and stream many audio and video codecs and file formats such as DVDs, VCDs and various streaming protocols. The most important functionality of VLC is the possibility to stream multimedia content over the network, which is useful for testing and evaluating the basic functionalities of WinJTAP.

### 5.1.2 Testing WinJTAP with VLC

Testing a WinJTAP interface with VLC gives us no quantitative information about performance. Thus, with observing the multimedia content, we can evaluate the efficiency and stability of WinJTAP subjectively. We tested WinJTAP with VLC in combination with the JTAPHub application, which was used for establishing P2P connections between group members. The tests have been accomplished using two different video stream bandwidths:

1. MPEG-1 video codec with a bandwidth of 224 kbps
2. MPEG-4 video codec (DivX) with Variable Bit Rate (VBR): max. ~2576 kbps

Our test peer group consisted of three peers (PCs) with the following configurations:

a) Intel Pentium 4 Processor, clock speed: 3.0 GHz, 1 GB RAM
b) Intel Pentium Mobile Centrino Processor, clock speed: 1.6 GHz, 1 GB RAM
c) Intel Pentium 4 Processor, clock speed: 1.7 GHz, 768 MB RAM

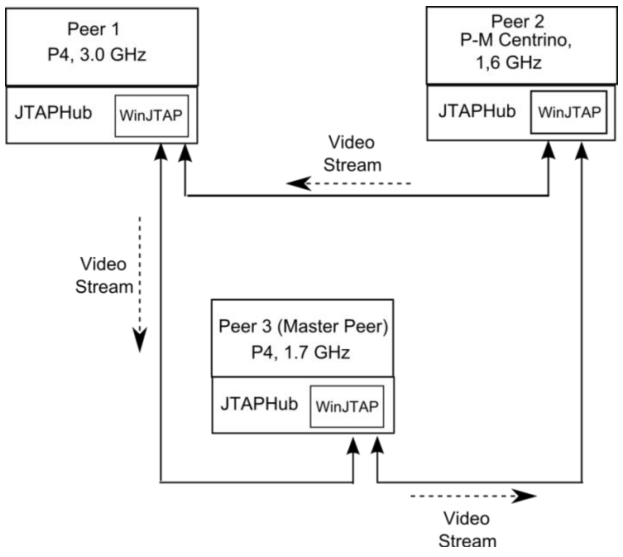


Figure 5.1 Configurations of Connected Peers in a P2P Group

At the beginning of our test, we created a P2P group with three members using the JTAPHub application. Further we started network video streaming using VLC from one peer to the other peers in the group. The video streams were sent to the virtual IP addresses of the TAP interface of group members. All peers, connected via JTAPHub, were able to receive a video stream over VLC without any delays or errors.
This first step in the evaluation process provided us the basic information about the functionality of WinJTAP. With help of this information, we could also conclude intuitively, that the achieved data throughput was sufficient for exchanging the video content. But this could not give us the precise information about WinJTAP's performance. For this reason, we had to use other testing methods (tools).

## 5.2 MGEN Application

### 5.2.1 **Overview**

One of the tools, which has been used for detailed performance analysis, is the Multi-Generator (MGEN) [7] software. MGEN is open source software, developed by the Naval Research Laboratory. It allows performing network performance tests and measurements using the possibility to generate UDP/IP traffic (real-time traffic patterns) and to send it to a given IP address. The received traffic can be also logged for analysis. The logged data gives us important information, such as the time of the packet reception or the sequence number of received packets, with whose help we are able to calculate the data throughput, packet loss rate and the variation in the communication delay (jitter). The possibility to generate sequence numbers for packets as well as having the time of reception was the main reason of using MGEN for performance evaluation of WinJTAP. For starting and controlling the simulations, MGEN uses script files. With the help of script files, we are able to control the data throughput at a given time point during the simulation. For example, we can start with initial data throughput (e.g. 1 Kbit/s) and then increase it by an arbitrary constant value during the test flow. As with the VLC test, we used JTAPHub for establishing P2P connections between test peers.

### 5.2.2 **Testing Concept**

Testing WinJTAP with MGEN has been accomplished by sending generated traffic from one sender to one receiver, without forwarding packets to the other peers (see Fig. 5.1).

The following configurations are used for testing:

1. Intel Pentium 4, clock speed: 3.0 GHz, 1 GB RAM
2. Intel Pentium Mobile Centrino, clock speed: 1.6 GHz, 1 GB RAM
3. Intel Pentium 4, clock speed: 1.7 GHz, 768 MB RAM

During the testing process, the bandwidth throughput was constantly increased by steps of ~2 MBit/ s after a given time interval.

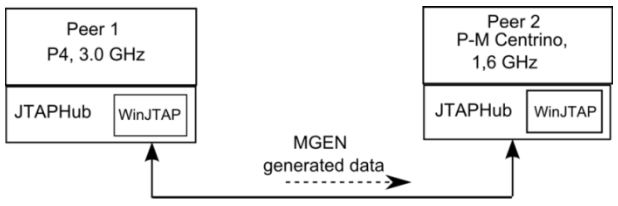


Figure 5.1 One possible Configuration of Peers during the Testing of WinJTAP with MGEN.

The process of testing WinJTAP with MGEN is described as follows:

a) Writing the MGEN script files for sender and receiver (See Fig. A.6.1 in the Appendix)
b) Starting the simulation with the MGEN scripts
c) Evaluation of results with help of the generated log files (see Fig. A.6.2 in the Appendix)

### 5.2.3 **Writing MGEN script files**

As described in the overview, MGEN uses script files for starting and controlling the network tests. MGEN script files are text files, which contain sequences of commands and events that are used for describing the generation of the network traffic. Those sequences (script lines) are parsed by MGEN. There are three main events/commands used for the generation of the network traffic:

- Transmission Event
- Reception Event
- Global Command

The scheduling of the commands and events at a given time point is achieved using the `<eventTime>` field, which accepts an arbitrary floating point number, a representation of the actual time, as argument. Other important fields used in MGEN script files are:

- `<flowId>:` Used for the identification of different flows
- `<procotol>:` Indicates the transport protocol, used for testing (only the UDP protocol is supported at the moment).
- `<address>` and `<port>`: Used for indication of source/destination addresses
- `<pattern [parameters]>:` Describes the art of the generation of network traffic together with pattern-parameters, which denotes the frequency and size of generated (UDP) messages. The mostly used pattern is the “PERIODIC” pattern, which generates fixed size messages (in bytes) at a constant rate (messages/second).

Important events are:

- “ON”/”OFF”: Used for starting/terminating the flow (used in combination with `<flowId>`).
- “SRC”/”DST”: Denote the source/destination for the generated flow (used in combination with `<address>` and `<port>` fields).

Example of MGEN script line is given as follows:

```
0.0 ON 1 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [500 1024].
```

See also Appendix, Fig. A.7.1 for more examples.

In the above example we can notice the described “events” and “fields”. This script line can be described as follows:

The UDP datagram flow (with flowId 1) is generated, using fixed size messages with the PERIODIC pattern, at the time point “0.0” from source, denoted with a port number “5001” to the destination, denoted with the combination of the IP address and port “10.3.0.2/5001”.

Important to mention are two parameters in the square brackets (“pattern section“), which control the actual data throughput of the generated flow. The first parameter sets the sending rate of the messages per second (e.g. 500 times per second) and the second parameter sets the size of the messages in bytes (e.g. 1024 bytes). In our MGEN script line example on page 28, the data is sent with throughput of ~ 4 MBit/sec. This can be easily calculated with the given parameters:

- 1024 Bytes * 8 = 8192 Bits.
- 8192 Bits * 500/sec = 4’096’000 Bits /sec.

See also Fig. A.7.1 in the Appendix.

## 5.2.4 **Starting the Simulation with MGEN**

MGEN is a shell application, which can be started from the command line using the executive file, “mgen.exe” delivered with the MGEN software. The execution can be accomplished using the combination of many parameters (command line syntax). For our tests we used the following parameters:

- `[input <scriptFile>]`: Used for reading from the MGEN script file.
- `[port <recvPortList>]`: Used for establishing the P2P connection on the given port number as argument.
- `[output <logFile>]`: Used for logging the results.

When no parameter for logging the data is used, MGEN logs automatically to the “stdout”. Starting the MGEN application with given parameters is accomplished, using the concatenation of the executable file and parameters. The following examples illustrate a practical application:

- Example 1: `mgen input scriptFile.txt`

  This command-line starts the MGEN simulation using the Script.txt file.

- Example 2: `mgen port 5003 output logFile.drc`

  This command line “listen” UDP traffic on port 5003 and logs the data to the “logFile.drc” file.

Once that a MGEN simulation is started, it will begin to generate the defined network traffic and sends it to the receiver using the appropriate destination IP address/port. The reception of the generated traffic will be logged. The generated log files give us important information for further (performance) evaluation of WinJTAP.

### 5.2.5 **Evaluation Process**

As with script files, we have also different events in the log file format. The following convention is used during the generation of log events:

```
<eventTime> <eventType> <event attributes>
```

The `<eventTime>` is in the form *Hour:Minute:Seconds* and indicates the occurrence time of an event.
The `<eventType>` denotes different events generated during the MGEN simulation.

Important event types are:

- RECV: Indicates the arrival of received generated message.
- RERR: Indicates the error at receiving the message.
- SEND: Indicates the transmitted message.

For our testing purposes the RECV event is important. It has the following format:

```
<eventTime> RECV flow> <flowId> seq> <sequenceNumber> src> <addr>
/<port> dst> <addr>/<port> sent><txTime> size><bytes>
[host><addr>/<port>] [gps><status>,<lat>,<long>,<alt>] [da-
ta><len>:<data>]
```

An example of a generated RECV log line looks as follows:

```
18:57:24.703125 RECV flow>1 seq>2 src>10.3.0.1/5001
dst>10.3.0.2/5001 sent>18:57:22.703125 size>1024
```

Here we can notice the time stamp of received messages (`<eventTime>`) and sent messages (`sent><txTime>`) fields, the sequence number of the messages, the source and destination address, as well as the message (UDP datagram) size in bytes. With help of the message size and the time of the reception, we are able to calculate the data throughput for the generated flow. The lost packet rate can be calculated with help of sequence numbers (`seq> <sequenceNumber>`). We can easily locate all missing packets at the receiver side using sequence numbers. Using time stamps (`<eventTime>` field) of the received and sent messages, we can also calculate the message delivery latency (jitter).

For detailed information about MGEN, refer to [7].

# 6. Test Results

As seen in Chapter 5, testing WinJTAP with MGEN gives us quantitative information about its performance. Considering the fact that our test environment consisted of explicitly software based applications (e.g. Java), a strong correlation between performance of WinJTAP and the peer performance (CPU, Memory and Chipset) is expected.
This correlation can be explained through the representation of the constantly exchanged data messages as Objects, which implicates actually the main cause of the difference in performance of WinJTAP in our test scenarios, which is the “Garbage Collector” (GC). GC is a mechanism, which scans periodically the memory space and removes the variables and objects, which are not in use, from the memory space, due to the performance and safety reasons.
Although Garbage Collectors have many advantages, they still require specific calculations and therefore can play an important role in achieving the performance for real time applications, such as multimedia applications.
We will discuss shortly, why the “Garbage Collector” has a big influence on the performance of WinJTAP when used in combination with JTAPHub.
As mentioned, JTAPHub uses serialization of read buffer content to create the Data Messages (Java Objects), which are stored in the memory space during the run-time of the JTAPHub application. Every time when some buffer content is read from a TAP device, JTAPHub makes a Java Object out of it. The generated Java Objects are not directly processed by the application. They are stored in a queue and therefore they are using some amount of memory during the time they are spending in the queue. Once they are consumed from the queue by the application, the reserved memory space will be cleared with help of the GC. During small bandwidth throughput, this mechanism is functioning perfect.
GC achieves to clear the memory space on time and there is no loss of data. On the other hand, sending the Data Messages with a higher bandwidth throughput makes it for the GC impossible to achieve to clear the data so fast, that the new Data Messages can reserve a memory space and this causes overloading in the queue and therefore Data Messages are lost. Of course, the speed of this mechanism depends on the following processes:

- Buffer content serialization
- Generation of objects
- Reservation of the memory space
- Data processing by the application
- Clearing of the memory space.

Those processes depend on the computing power (CPU, RAM access time, Chipset). Computers with high CPU frequency and RAM speed will achieve faster generation of objects and cleaning of the memory space. Processing of the data will also be faster.
In the Fig. 6.1 we can see the clear difference between different configurations (different chipset architecture, different CPU frequency). The first two configurations (P4, 1.7 GHz with Pentium M 1.6 GHz and P4 3.0 GHz with Pentium M 1.6 GHz) have achieved quite similar bandwidth throughput, due to similar technical characteristics. On the other hand, the third combination of Peers (P4, 3 GHz with P4, 1.7 GHz) has a clear difference in the bandwidth throughput.

The more powerful pair of peers had also a lower dropped packet rate at higher bandwidth rate than the other pairs. The maximum achieved bandwidth throughput was **~19.6 MBit/s**. This result depends on used test environment and can differ in the case of using other test environments (P2P Application, Measurement software). In our case, WinJTAP was tested with a JTAPHub P2P application, which was not performance optimized. The achieved bandwidth throughput and packet loss rate can be different using WinJTAP with MM due to a different architecture.

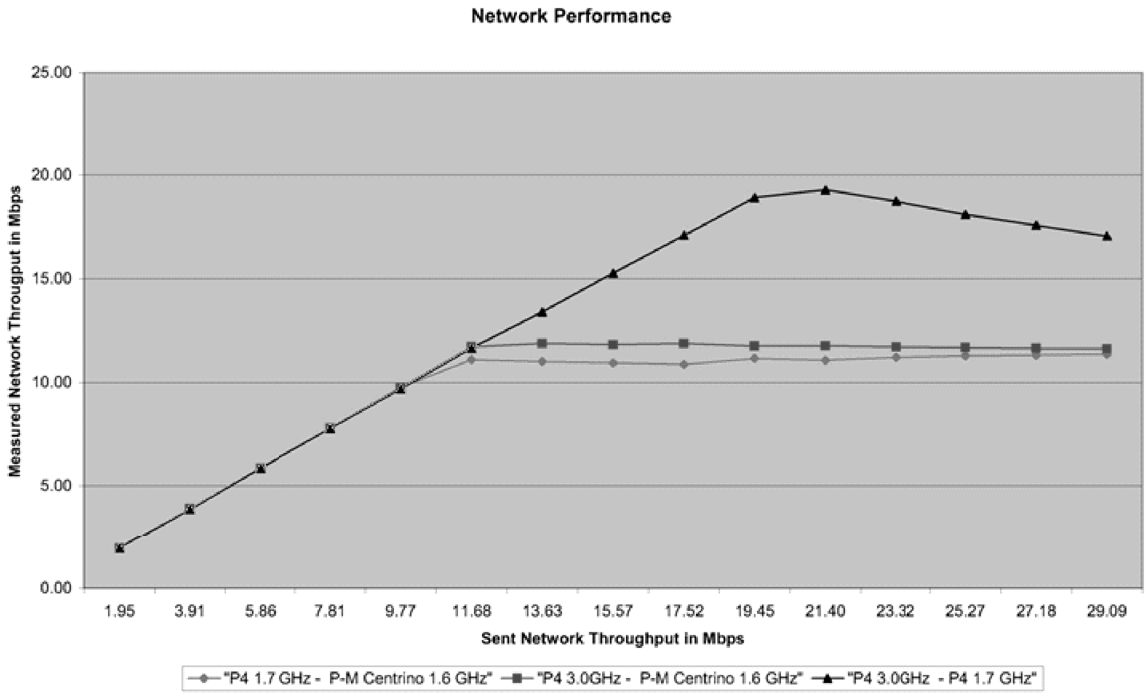


Figure 6.1 Network Performance of the WinJTAP as a Result of the MGEN Simulation

Observing the diagram in the Fig. 6.1 we can also notice a very interesting behaviour of the curve with the P4 3 GHz with P4 1.7 GHz peer-combination.
Here we can notice that the curve falls slightly after the maximum bandwidth throughput has been reached. This behaviour is a consequence of the so called "congestion control effect", which occurs using TCP communication in the JTAPHub application (Socket communication). At the TCP communication level, the sender always receives acknowledgments from the receiver, to ensure that every packet is correctly received. When a packet is dropped for some reason (e.g. transmission error or overloading), the sender will not receive an acknowledgement and will know that traffic congestion occurred. To mitigate the congestion, the sender will automatically reduce the congestion window, which will actually affect the throughput, since the congestion window controls the throughput directly. For more details about TCP congestion control refer to RFC 2581 in [11].

Reaching the limit of the bandwidth throughput can also be noticed when observing the packet loss rate (see the diagram in Fig. 6.2). At some point, the packet loss rate will rapidly grow up to the maximum value, which gives a clear sign that the maximum possible bandwidth throughput has been reached (see Fig 6.1, the curve with the configuration: P4, 3 GHz and P4, 1.7 GHz).

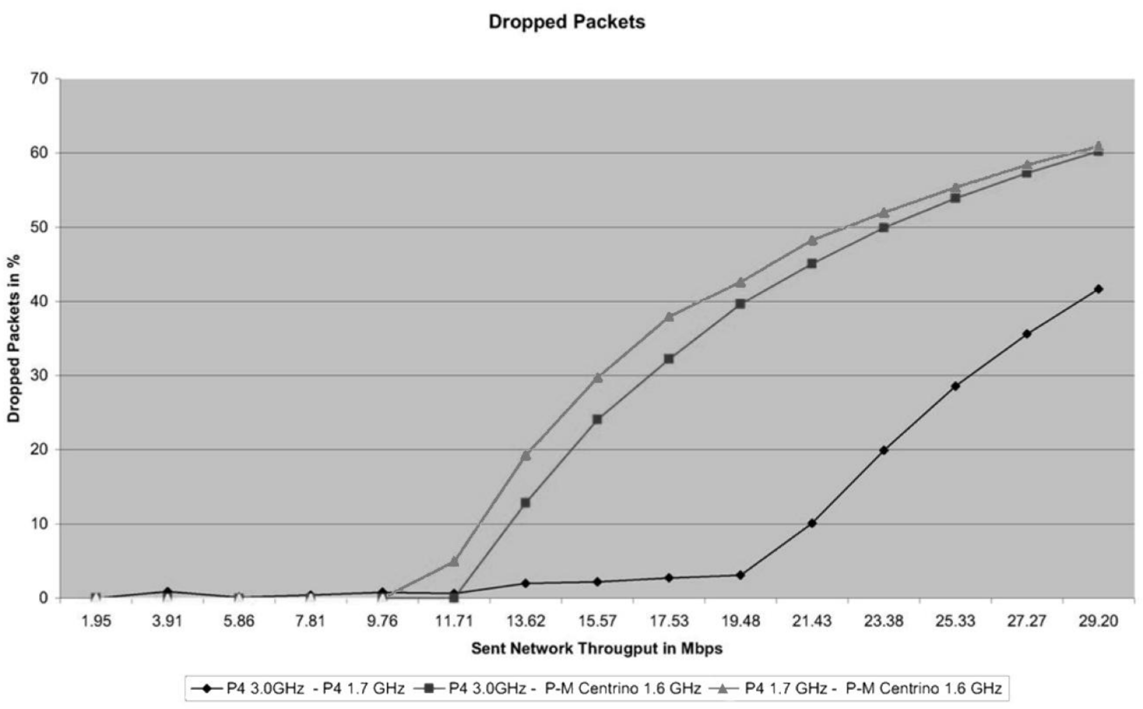


Figure 6.2 Dropped Packets during Testing WinJTAP with MGEN

# 7. Conclusions and Outlook

In this computer science project, we implemented and evaluated the WinJTAP interface. Further, we implemented a test application, which was used for the evaluation of WinJTAP. We also analyzed the TAP-Win32 device driver architecture in order to understand the communication flow of TAP in Win32. Finally we evaluated the functionality and performance of WinJTAP with the help of JTAPHub, VLC and MGEN applications.

The results clearly show that WinJTAP can be used for the Multicast Middleware (MM), regarding the performance (~19 MBit/s), dropped packets and the test environment, which was not performance optimized. The future work should focus on the optimization of the WinJTAP interface (optimization of buffer-management) and also on the integration and evaluation of WinJTAP with the MM. Decreasing the “Garbage Collector effect” on the MM side would also be one important issue for our future work. Since the performance optimization of our test environment (JTAPHub) would not contribute to the real performance of WinJTAP, there is no indication for any further improvements by it.

Another important issue would also be the migration (integration) of WinJTAP to the Windows Vista. Windows Vista migration would be desirable due to standardization reasons.

# 8. Bibliography

[1] **D. Milic, M. Brogle, T. Braun.** *Video Broadcasting using Overlay Multicast.* University Bern. s.l.: Seventh IEEE International Symposium on Multimedia (ISM 2005), Irvine, California, USA, December 12 - 14, 2005, pp. 515-522, IEEE Computer Society Press, ISBN 0-7695-2489-3.

[2] EuQoS Project. http://www.euqos.eu/.

[3] Intel. http://resource.intel.com.

[4] Java API Documentation. http://java.sun.com/j2se/1.4.2/docs/api/.

[5] **Krasnyansky, Maxim.** Universal TUN/ TAP. http://vtun.sourceforge.net/tun/.

[6] **M. Brogle, D. Milic.** *EuQoS Multicast Middleware: Basic Architecture Overview and Concepts.* s.l.: Technical Report, IAM-05-002, Institute of Computer Science and Applied Mathematics.

[7] MGEN, Naval Research Laboratory. http://cs.itd.nrl.navy.mil/work/mgen/.

[8] Microsoft Developer Network (MSDN). http://msdn2.microsoft.com/.

[9] PCAUSA. http://www.pcausa.com/resources/.

[10] **Sheng Liang, Sun Microsystems, Inc.** Java Native Interface, programmers guide and specification.

[11] The Internet Engeneering Task Force (IETF). http://www.ietf.org/.

[12] VideoLAN Project. http://www.videolan.org/.

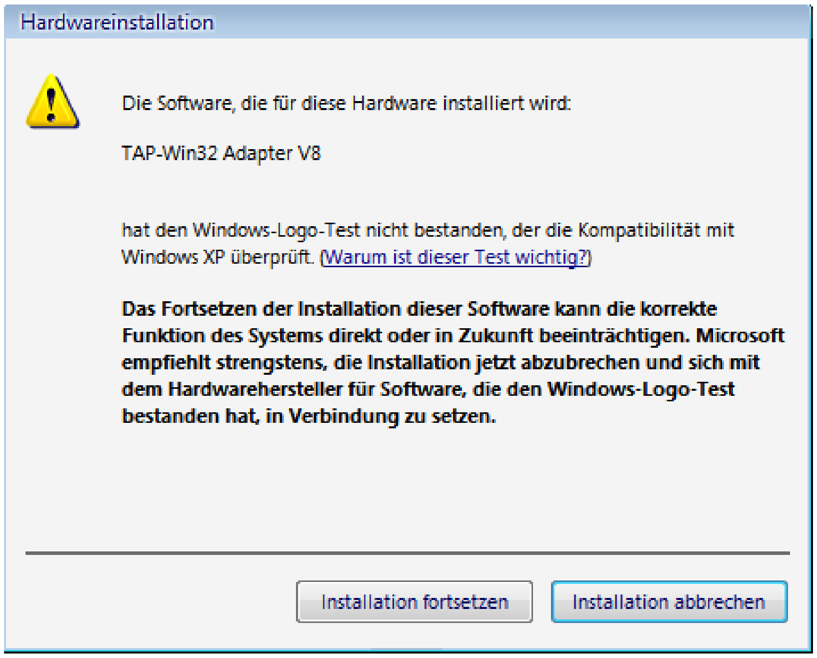
[13] **Wilson, Damion K.** Cipe-Win32. http://cipe-win32.sourceforge.net/.

[14] **Yonan, James.** OpenVPN. http://www.openvpn.net/.

# A. Appendix

## A.1 Installing and Configuring TAP on Win32

Installation and configuration of the TAP interface on Win32 requires installation of OpenVPN [14]. A Windows version of OpenVPN can be installed from the self-installing exe file found on the OpenVPN's download page. During the installation, the screen for installing the TAP-Win32 driver will be shown as seen in Fig. A.1.1.



Die Software, die für diese Hardware installiert wird:

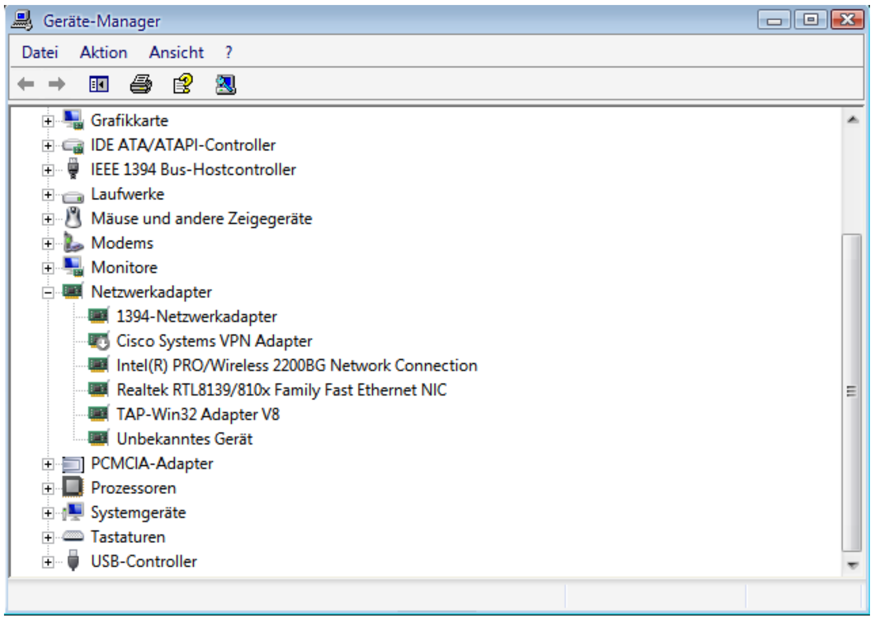
TAP-Win32 Adapter V8

hat den Windows-Logo-Test nicht bestanden, der die Kompatibilität mit Windows XP überprüft. (Warum ist dieser Test wichtig?)

**Das Fortsetzen der Installation dieser Software kann die korrekte Funktion des Systems direkt oder in Zukunft beeinträchtigen. Microsoft empfiehlt strengstens, die Installation jetzt abzubrechen und sich mit dem Hardwarehersteller für Software, die den Windows-Logo-Test bestanden hat, in Verbindung zu setzen.**

Figure A.1.1 Installation of the TAP-Win32 Driver

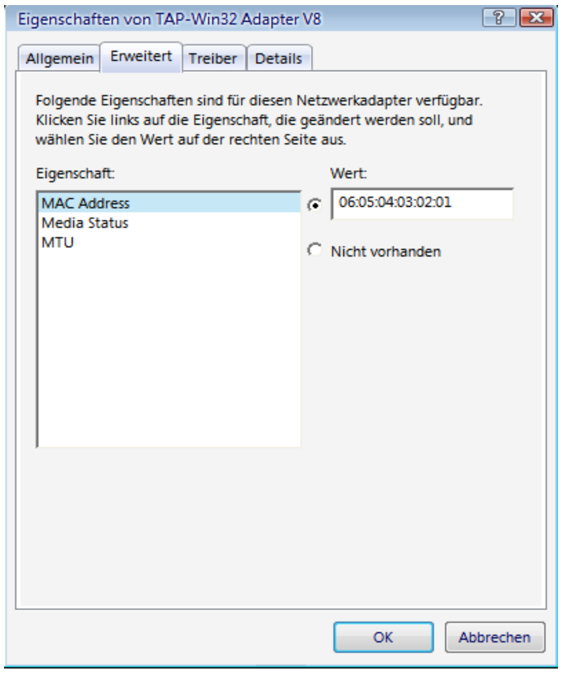
After the installation, the TAP interface driver will be treated in the OS as a normal Ethernet device and will have all functionalities of a physical Ethernet device (see Fig. A.1.2).



Grafikkarte
IDE ATA/ATAPI-Controller
IEEE 1394 Bus-Hostcontroller
Laufwerke
Mäuse und andere Zeigegeräte
Modems
Monitore
Netzwerkadapter
1394-Netzwerkadapter
Cisco Systems VPN Adapter
Intel(R) PRO/Wireless 2200BG Network Connection
Realtek RTL8139/810x Family Fast Ethernet NIC
TAP-Win32 Adapter V8
Unbekanntes Gerät
PCMCIA-Adapter
Prozessoren
Systemgeräte
Tastaturen
USB-Controller

Figure A.1.2 Tap Interface as a Network Device in Win32

In the properties section of the TAP device driver, the MAC can be set address manually (see Fig. A.1.3).



Folgende Eigenschaften sind für diesen Netzwerkadapter verfügbar. Klicken Sie links auf die Eigenschaft, die geändert werden soll, und wählen Sie den Wert auf der rechten Seite aus.

Eigenschaft:
MAC Address
Media Status
MTU

Wert:
[✓] 06:05:04:03:02:01
Nicht vorhanden

Figure A.1.3 Mac Address Settings

## A.2 OpenVPN’s TAP interface architecture on Win32

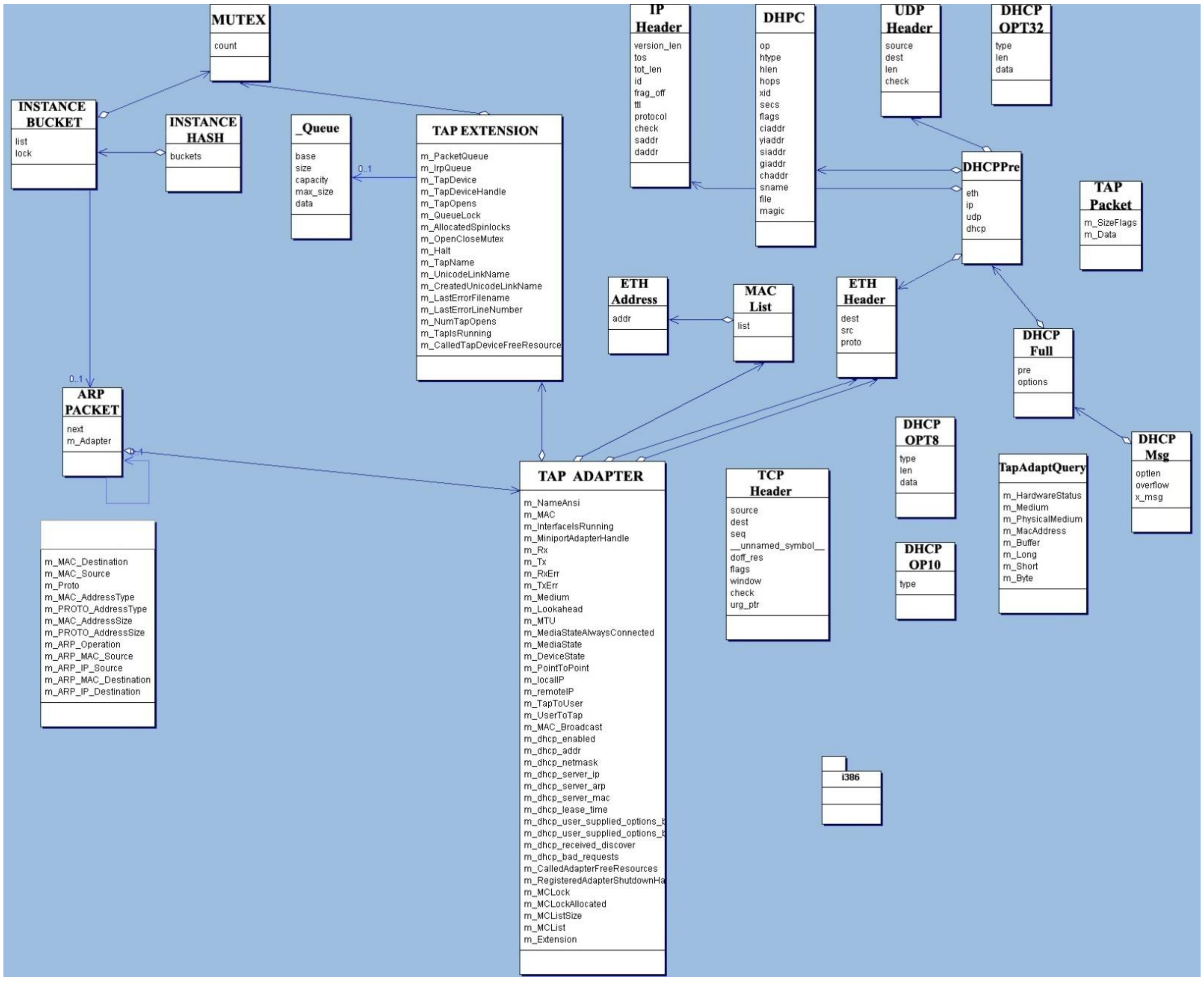


Figure A.2.1 OpenVPN’s TAP Device Architecture in Win32 described with UML language

## A.3 Overlapped Structure and Device I/O Control Functions

```
typedef struct _OVERLAPPED {
    ULONG_PTR Internal;
    ULONG_PTR InternalHigh;
    union {
        struct {
            DWORD Offset;
            DWORD OffsetHigh;
        };

        PVOID Pointer;
    };

    HANDLE hEvent;
} OVERLAPPED, *LPOVERLAPPED;
```

Source :  http://msdn2.microsoft.com/

Figure A.2.1 Overlapped Structure

```
BOOL WINAPI ReadFile(
  HANDLE hFile,
  LPVOID lpBuffer,
  DWORD nNumberOfBytesToRead,
  LPDWORD lpNumberOfBytesRead,
  LPOVERLAPPED lpOverlapped
);
```

Source: http://msdn2.microsoft.com/

Figure A.2.2 ReadFile I/O Function

```
BOOL WINAPI WriteFile(
  HANDLE hFile,
  LPCVOID lpBuffer,
  DWORD nNumberOfBytesToWrite,
  LPDWORD lpNumberOfBytesWritten,
  LPOVERLAPPED lpOverlapped

);
```

Source: http://msdn2.microsoft.com/

Figure A.2.3 WriteFile I/O Function

```
HANDLE WINAPI CreateFile(
  LPCTSTR lpFileName,
  DWORD dwDesiredAccess,
  DWORD dwShareMode,
  LPSECURITY_ATTRIBUTES lpSecurityAttributes,
  DWORD dwCreationDisposition,
  DWORD dwFlagsAndAttributes,
  HANDLE hTemplateFile
);
```

Source: http://msdn2.microsoft.com/

Figure A.2.4 CreateFile I/O Function

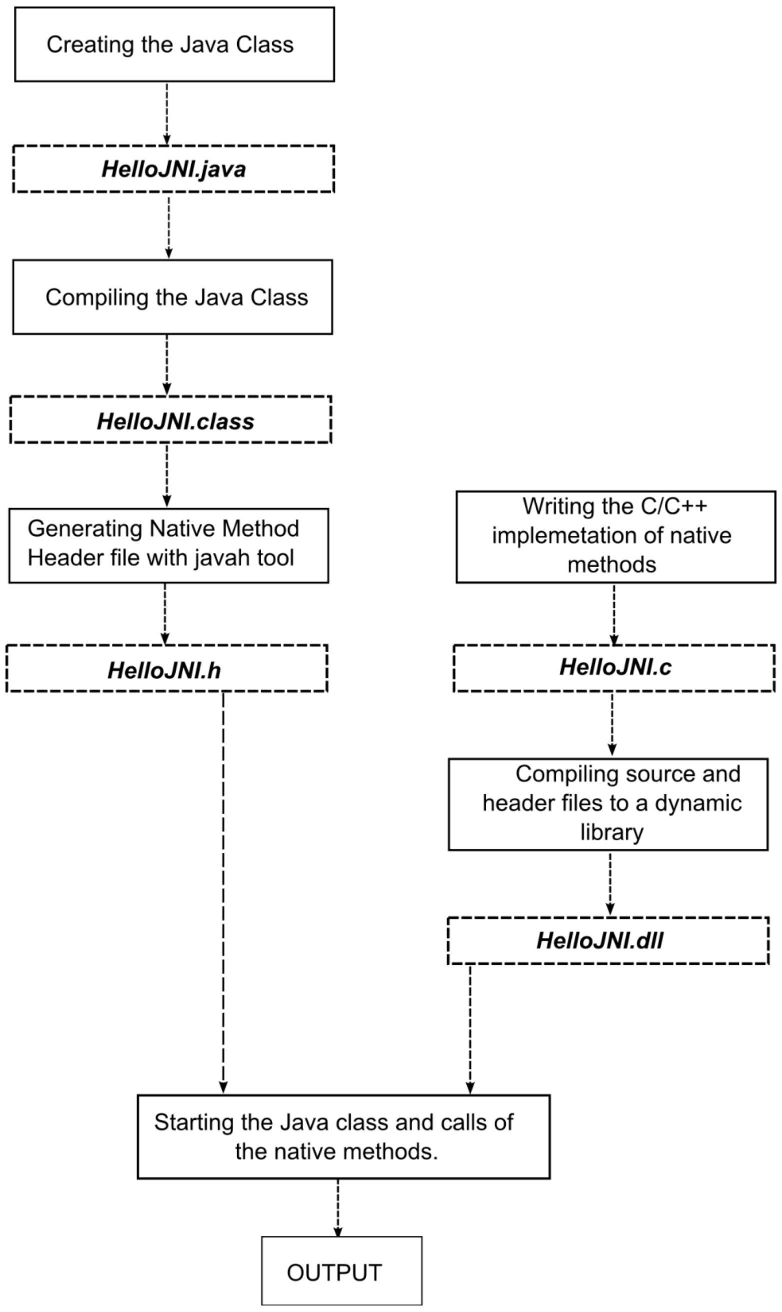
```
BOOL WINAPI DeviceIoControl(
  HANDLE hDevice,
  DWORD dwIoControlCode,
  LPVOID lpInBuffer,
  DWORD nInBufferSize,
  LPVOID lpOutBuffer,
  DWORD nOutBufferSize,
  LPDWORD lpBytesReturned,
  LPOVERLAPPED lpOverlapped
);
```

Source: http://msdn2.microsoft.com/

Figure A.2.5 DeviceIoControl Function

For further information about I/O functions and Overlapped Structure refer to [8].

## A.4 Java Native Interface (Process of creation and starting the Application using JNI)



Creating the Java Class
HelloJNI.java
Compiling the Java Class
HelloJNI.class
Generating Native Method Header file with javah tool
HelloJNI.h
Writing the C/C++ implemetation of native methods
HelloJNI.c
Compiling source and header files to a dynamic library
HelloJNI.dll
Starting the Java class and calls of the native methods.
OUTPUT

Figure A.3.1 Process of Writing and Running the C Program from Java using JNI

## A.5 Example of JNI Functions written for Communicating with TAP

```
JNIEXPORT jint JNICALL Java_TapInterface_openNewInterface
  (JNIEnv *env, jclass class)
{

     jint handle;

     if(tap_open(&handle, "tap-bridge") != 0)
     {
      return -1;
     }

     return handle;
}
```

Figure A.4.1 JNI Function for Opening TAP Interface

```
JNIEXPORT jint JNICALL Java_tunTapInterface_TapInterface_readPacket
  (JNIEnv *env, jclass class, jint descriptor, jbyteArray bufferArray) {

      int len=(*env)->GetArrayLength(env,bufferArray);
      int readBytes;
      char *data;

      jbyte* bytes=(*env)->GetByteArrayElements(env,bufferArray,NULL);
      readBytes = tap_read(descriptor,&data,len);

      if (readBytes > 0)
      {
         memcpy(bytes,data,readBytes);
         release_buffer(descriptor,data);
        (*env)->ReleaseByteArrayElements(env,bufferArray,bytes,0);
      }

      else if (readBytes < 0)
      {
         throwTunTapException(env,"error reading the packet");
         return -1;
      }
      else
      {
         return readBytes;
      }

}
```

Figure A.4.2 JNI Function, which calls a C Function for Reading from TAP Interface

```
JNIEXPORT void JNICALL Java_TapInterface_sendPacket
(JNIEnv *env, jclass class, jint descriptor, jbyteArray bufferArray, jint
length)
{

      jbyte* bytes=(*env)->GetByteArrayElements(env,bufferArray,NULL);

      int writtenBytes= tap_write(descriptor,bytes,length);

     (*env)->ReleaseByteArrayElements(env,bufferArray,bytes,0);

      if (writtenBytes == -1)
      {
            throwTunTapException(env,"error writing the packet");
            return -1;
      }

      else
      {
          return writtenBytes;
      }

}
```

Figure A.4.3 JNI Function which calls a C Function for Writing to TAP Interface

```
JNIEXPORT void JNICALL Java _TapInterface_closeInterface
 (JNIEnv *env, jclass class, jint descriptor)
{
    tap_close((HANDLE)descriptor);
}
```

Figure A.4.4 JNI Function for Closing the TAP Interface

```
private native static int openNewInterface(byte[] interfaceAddress) throws
IOException;

private native static int readPacket(int descriptor, byte[] buffer) throws
IOException;

private native static void sendPacket(int descriptor, byte[] buffer) throws
IOException;

private native static void closeInterface(int descriptor) throws
IOException;
```

Figure A.4.5 Declarations of Native Methods in Java, used for Calling TAP related Functions

## A.6 JTAPHub Architecture

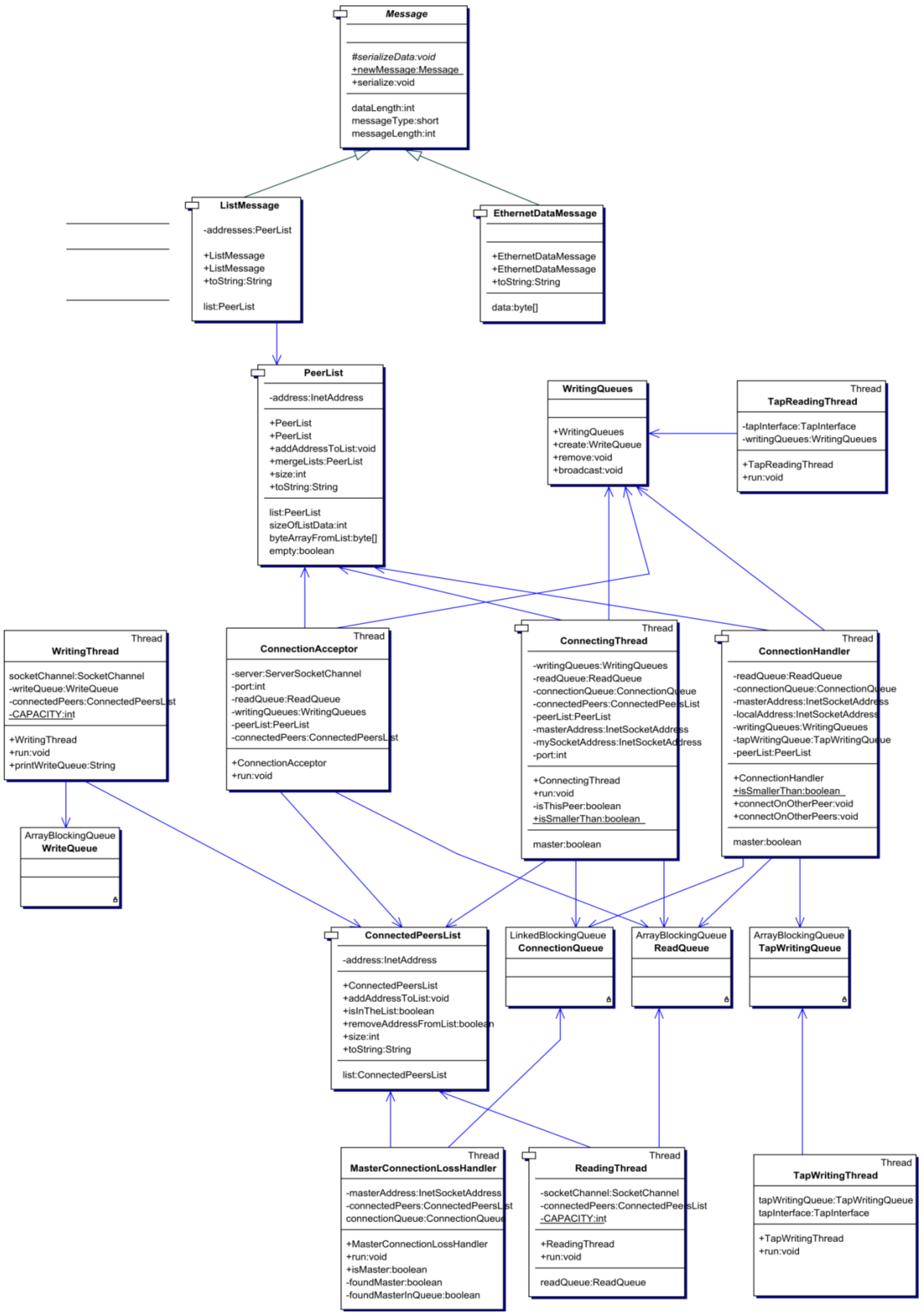


Figure A.6.1 Class Diagram of implemented JTAPHub Interface

## A.7 MGEN Script and Log File Examples

```
# MGEN script begins here


#4 Mbit/s
0.0 ON 1 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [500 1024]
10.0 OFF

#6 Mbit/s
0.0 ON 2 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [750 1024]
10.0 OFF 2

#8 Mbit/s
11.0 ON 2 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [1000 1024]
21.0 OFF 2

#10 Mbit/s
22.0 ON 3 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [1250 1024]
32.0 OFF 3

#12 Mbit/s
33.0 ON 4 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [1500 1024]
43.0 OFF 4

#16 Mbit/s
44.0 ON 4 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [1750 1024]
54.0 OFF 4

#18 Mbit/s
55.0 ON 5 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [2000 1024]
65.0 OFF 5

33.0 ON 1 UDP SRC 5001 DST 10.3.0.2/5001 PERIODIC [2250 1024]
43.0 OFF 1
```

Figure A.7.1 Example of MGEN Script File

```
18:57:24.703125 RECV flow>1 seq>2 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.703125 size>1024
18:57:24.718750 RECV flow>1 seq>3 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.718750 size>1024
18:57:24.718750 RECV flow>1 seq>4 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.718750 size>1024
18:57:24.718750 RECV flow>1 seq>5 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.718750 size>1024
18:57:24.718750 RECV flow>1 seq>6 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.718750 size>1024
18:57:24.734375 RECV flow>1 seq>7 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.734375 size>1024
18:57:24.734375 RECV flow>1 seq>8 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.734375 size>1024
18:57:24.734375 RECV flow>1 seq>9 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.734375 size>1024
18:57:24.734375 RECV flow>1 seq>10 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.734375 size>1024
18:57:24.750000 RECV flow>1 seq>11 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.750000 size>1024
18:57:24.7500RECV flow>1 seq>12 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.750000 size>1024
18:57:24.750000 RECV flow>1 seq>13 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.750000 size>1024
18:57:24.750000 RECV flow>1 seq>14 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.750000 size>1024
18:57:24.765625 RECV flow>1 seq>15 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.765625 size>1024
18:57:24.765625 RECV flow>1 seq>16 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.765625 size>1024
18:57:24.765625 RECV flow>1 seq>17 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.765625 size>1024
18:57:24.765625 RECV flow>1 seq>18 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.765625 size>1024
18:57:24.781250 RECV flow>1 seq>19 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.781250 size>1024
18:57:24.781250 RECV flow>1 seq>20 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.781250 size>1024
18:57:24.796875 RECV flow>1 seq>21 src>10.3.0.1/5001 dst>10.3.0.2/5001
sent>18:57:22.781250 size>1024
```

Figure A.7.2 Example of generated Log File with MGEN Application.