Running virtualized native drivers in User Mode Linux *

V. Guffens G. Bastin
Centre for Systems Engineering and Applied Mechanics (CESAME)
Université Catholique de Louvain, Belgium
{*guffens,bastin*}*@auto.ucl.ac.be*

Abstract

A simulation infrastructure for wireless network emulation based on User Mode Linux and on the virtualisation of the hostap driver is proposed. The interconnection of these components is first described and the architecture of the resulting network emulator is explained. Two practical applications are then detailed : the testing of an implementation of the AODV routing protocol in a highly realistic environment and the study of the interactions between the hostap driver and the card it drives.

1 Introduction

User Mode Linux (UML [1]) has proven to be very useful for kernel debugging ([9], chap. 4), for the implementation of new functionalities in the kernel and as a testing and teaching tools. The Openswan developers, for instance, report in [7] the use of UML as a testing and development tools for their project. It is also used in [2] for network protocol testing with the VNUML project. UML is also used to implement web hosting solutions, honeypots and redundant services.

Some customs drivers exist in UML that provide, as an example, network connectivity. One of the existing Ethernet driver, for instance, works by opening a *tap* interface on the host side and by presenting this interface at the UML kernel side as the usual *net_device* structure with the suitable interfacing functions. From the point of view of the user land tools such as *ifconfig* or *ip*, this interface can therefore be manipulated as any other real Ethernet interface would.

*This paper presents research results of the Belgian Programme on Interuniversity Attraction Poles, initiated by the Belgian Federal Science Policy Office. The scientific responsibility rests with its author(s).

However, these custom UML drivers are different from the native Linux drivers and the benefits mentioned above are therefore lost. Those benefits could nevertheless be recovered if the virtualisation process was carried on at a lower level. In this paper, we describe how this process can be successfully achieved by implementing a new bus, that we call *netbus*, which implements the functionalities found at the PCI level. This new bus allows for inserting native PCI Linux driver inside UML providing that some piece of code exists to emulate the hardware device that the driver is manipulating. To this end, the implementation of a software 802.11 card that can be operated by the *hostap* [6] driver is presented.

These software cards are then connected to each other trough a network server that we have developed (sources available at [3]) in QT/C++ and which provides a physical layer emulation as well as a graphical tool to represent the virtual machines in a 2-dimensional world (See figures later in the text). The advantages of such a system are as follows :

- It provides a complete wireless network emulator which is highly realistic and which can be used to test and develop new wireless related protocols. We report the implementation of the name resolution manet draft using this system in [4]. In this paper, we focus on the testing of the AODV multihop adhoc routing protocol. The implementation under investigation is the NIST kernel AODV module [5]. This implementation has been chosen because users from a wifi community [8] have reported random failures in the routing after many hours of operation. Diagnosing the cause of the failure on site might be a real nightmare and the solution is usually to reboot the device immediately. Reproducing the failure in the emulator might therefore be very valuable and help in development of such

citizen networks.

- It can be used as a teaching tool to understand how a driver works.

In Section 2, the architecture of the emulator is first described and in Section 3 and 4 it is shown how it can be used to efficiently debug the AODV routing protocol and to help in understanding the interactions between a driver and the hardware it drives. Some related works are mentioned in Section 5 and the conclusion is given in Section 6

2 Emulator architecture

As described in the introduction, the main components used in the emulator are the User Mode Linux kernel code and a wireless driver. For this work, we chose the `hostap` driver [6] because it supports different kind of hardware and its hardware dependent code is therefore neatly separated. Furthermore, the `hostap` driver may be setup as an access point and supports software-based encryption. The advantages of the Linux kernel are, among others, availability of the source code, availability of advanced networking features, increasing use in embedded network devices and huge amount of networking related softwares. In the context of this paper, the availability of a stable User Mode port is of course essential.

Linking the `hostap` driver with the User Mode kernel requires the resolution of all symbols from the kernel's exported symbol table which is not possible as User Mode Linux does not contain any bus implementation. Unresolved symbols, whose names are self-explanatory are for instance : `pci_enable_device`, `writew` and `readw`.

The first component that had to be implemented was therefore a simple bus suitable to export the required symbols for the `hostap` driver to be inserted in the kernel. The modification of the `hostap` driver is minimal and consists in replacing unresolved symbols with the new ones, along with some minor modifications due to the simpler implementation of our bus compared to a standard `PCI` bus. We refer to this bus as "netbus" as its primary use is to connect a network device to UML. The functionalities of this bus, sending and receiving data as well as transmitting interrupt signals are implemented with TCP connections. Once the `hostap` driver is successfully linked into the kernel, it must of course act on a device which also has to be emulated.

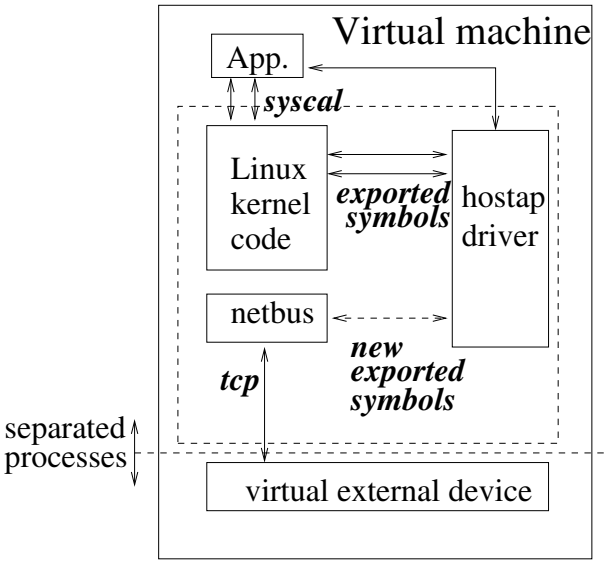

Figure 1: Interconnection of the driver with the UML kernel

2.1 Interconnection of the components

As any other device, our emulated wireless card has to be connected or "virtually plugged" into the UML via netbus. This situation is depicted in Fig. 1 showing the different interconnections existing between the components introduced above. The emulated wireless card runs as a different process, which is the emulator itself. This core program is written in QT/C++ and also implements the physical layer emulation as well as a visualisation system.

As mentioned above, netbus is built with TCP connections and the core emulator is therefore written as a TCP server.

The architecture of the emulator is depicted in Fig. 2 where it can be seen that the `hostap` driver is represented as a TCP client which is implemented on top of netbus. When the driver tries to register itself, it calls a function `netbus_register_device` which tries to bind to the emulator server socket. If the operation is successful, a new object implementing the virtual wireless card is instantiated in the emulator. The visualisation system displays an icon representing the mobile node with a surrounding circle which represents the reachability zone of the card (See more details later in the text).

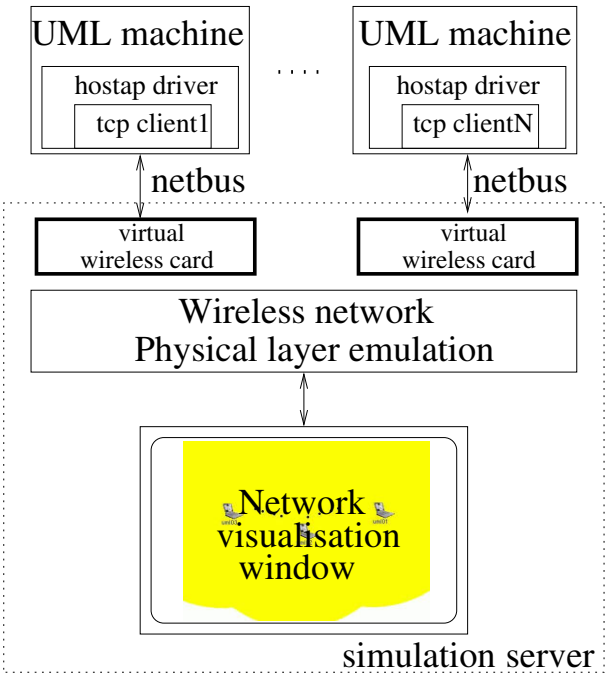

Figure 2: Architecture of our wireless emulator

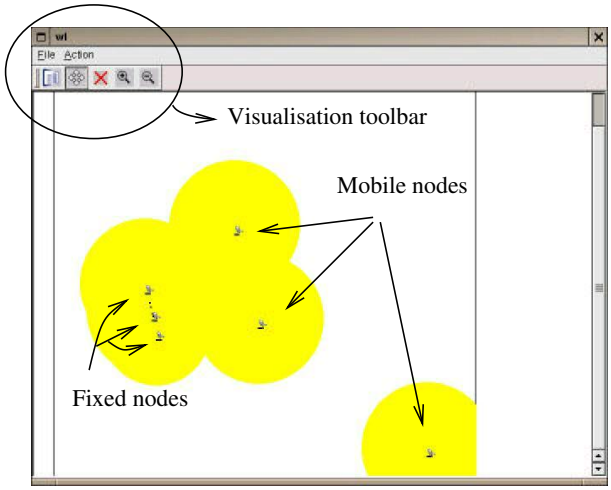

Figure 3: The setup used for testing the AODV protocol

2.2 Netbus implementation

As mentioned previously, the connection between the emulated card and the driver is realised with a TCP connection. Using TCP makes it possible to run multiple UML machines on different hosts. The simulation can therefore be distributed among multiple CPU's. The TCP protocol has been chosen for its reliability. Indeed, it is not desirable to introduce uncertainty in the delivery of interrupt and data as one would like to control in a deterministic fashion if packet drops occur or not.

In our prototype, two different TCP connections are used : one connection is used for data transmission while the other is only used for interrupt emission.

The client socket corresponding to the interrupt line at the UML side is configured in ASYNC mode and the associated SIGIO signal is registered in UML as an interrupt signal associated with the wireless card. The data line operates in blocking mode. This is necessary as, from the UML point of view, the read/write operation has to be "atomic" and must be completed entirely before the UML may continue to execute. Every command sent on the data line is therefore followed by a blocking `recv` call which waits for an acknowledgement from the card and synchronises the kernel with the card.

As packets are transmitted by the `hostap` driver by a series of `writew` command (programmed IO), the synchronisation mode described above induces a high latency in packet transmission. Therefore, an extra feature has been implemented in netbus which permits to send a block of "len" bytes of data in a single blocking operation. In some sense, this feature is analogous to a DMA in a real hardware.

3 A testbed environment

With the spreading of roof-top wifi networks, dedicated devices such as the *meshbox* and dedicated distributions such as *OpenAP* started to appear. Basically, these devices should be configured as AODV routers, installed at the right position and forgotten. Unfortunately, stability issues, interoperability problems or unexpected circumstances often prevent such an ideal situation to occur. In this Section, we report our attempt to reproduce a typical breakdown in a multihop ad-hoc network.

The setup is shown in Fig. 3 and a zoom on the fixed nodes is depicted in Fig. 4. A video captured during the experiment can be downloaded at [3]. The machine named $UMLx$ received and IP adresse of 192.168.0.x.

Obviously, this setup puts the system at loads and AODV messages are printed on the nodes console as soon as other nodes are coming in their reachability zone (as depicted by the yellow circles in Fig. 3). After letting the system run for a while, it was then found that a ping from $UML6$ to $UML3$ would only receive a single reply while a ping from $UML3$ to $UML6$ would receive no reply at all.

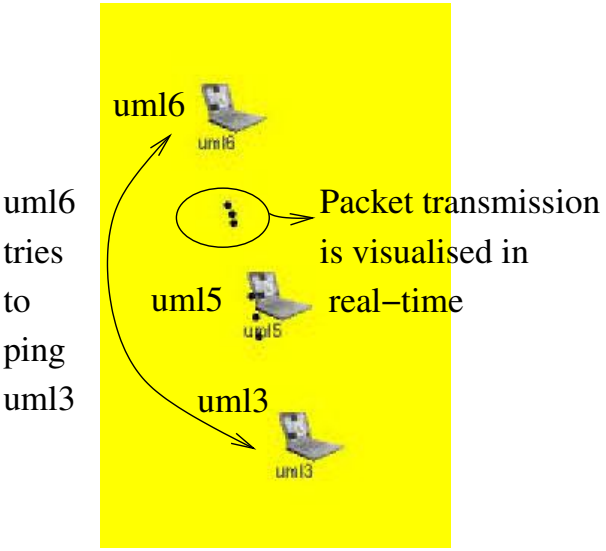

Figure 4: A zoom on the fixed nodes

A first tool at our disposal to understand what is happening is the use of the *tcpdump* packet capture program and the *ethereal* analyser. The trace acquired at $UML5$ is shown in Fig. 5. It can be seen in lines 35-38 that the ping request is arriving at $UML5$ and forwarded toward $UML3$ as is the first ping reply in the reversed direction. Lines 45-47 show the AODV hello messages with a lifetime of 3 seconds. The next trace, acquired at $UML03$ is shown in Fig. 6 with the Ethernet addresses where one can see the same sequence of two ping requests with one ping reply. However, the destination Ethernet address of the ping reply now comes as a surprise as it corresponds to the hardware address of $UML6$ showing that $UML3$ tries to bypass $UML5$. A look at Fig. 7 showing the AODV routing table of $UML6$ and $UML3$ reveals that $UML6$ is present in the routing table of $UML3$ while the reverse is not true. Indeed, it can be seen on the last line of Fig. 6 that $UML3$ can receive the hello messages from $UML6$.

A quick check revealed that the transmission power of $UML3$ had been modified at the beginning of the simulation and that its transmission range was therefore smaller than the transmission range of $UML6$. The first ping request was creating a reverse route on its way toward $UML3$ and the first reply was therefore reaching $UML6$. After the first hello message from $UML6$, the routing table in $UML3$ was modified and the ping replies were sent directly to $UML6$ which could not receive them because of the limited transmission power of $UML3$.

This practical example that actually occurred in the simulator is in fact widespread in practice and had been reported many times (for instance in [8]). It occurs notably when an AODV router is setup with a high power transmission while people trying to connect to it use a regular laptop card with smaller power transmission. The realistic physical transmission model used in the simulator allowed for recovering a typical situation often encountered in practice while the use of UML allowed for using conventional tools such as *tcpdump* for the debugging. This latter point might be of great practical interest for evaluating wireless oriented distribution before deploying them in the field.

4 A teaching tool

Although the results presented in the previous section did require an ergonomic simulation environment and a realistic enough physical layer emulation, the same results could have been obtained without the hassle of emulating in software the behaviour of a real hardware card. It would have been sufficient to code a UML specific driver just like the already existing tun/tap Ethernet driver. This would require, however, the complete rewriting of the user-land interface, including the wireless extension. In this section, we present some results that specifically exploit the virtualisation of the wireless driver.

Indeed, with the help of the emulated hardware card, it is very easy to obtain detailed information about the internal mechanisms involved in the operation of the card. For many programmers, the interactions between the driver and the device remain a *terra incognita* that we may easily explore with our emulator. Fig. 8 and Fig. 9 show two different traces that can easily be obtained from the emulator software log file. The trace of the Fig. 8 can be read as follows :

1-3 The driver sets the parameter for a future command to $0x93c$ and send the command CMDCODE_ALLOC which interprets the given parameter as the size of the segment to allocate

4-5 Internally, the driver sets the event register bit corresponding to the memory allocation event and writes the ID of the allocated frame at the expected memory location

7-9 The events are acknowledged by the driver

This initialisation phase allocates memory regions referred to by their frame ID to be used later for packet transmission and reception. This is shown in Fig. 9 which display a trace corresponding to the transmission of a data frame. The *netbus_send* com-

```
35 8.502093    192.168.0.6           192.168.0.3           ICMP     Echo (ping) request
36 8.502277    192.168.0.6           192.168.0.3           ICMP     Echo (ping) request
37 8.537036    192.168.0.3           192.168.0.6           ICMP     Echo (ping) reply
38 8.537144    192.168.0.3           192.168.0.6           ICMP     Echo (ping) reply
45 9.581531    192.168.0.3           255.255.255.255       AODV
                RREP D: 192.168.0.3 O: 192.168.0.3 Hcnt=0 DSN=1 Lifetime=3000
46 9.999939    192.168.0.5           255.255.255.255       AODV
                RREP D: 192.168.0.5 O: 192.168.0.5 Hcnt=0 DSN=1 Lifetime=3000
47 10.336736   192.168.0.6           255.255.255.255       AODV
                RREP D: 192.168.0.6 O: 192.168.0.6 Hcnt=0 DSN=2 Lifetime=3000
60 12.386353   192.168.0.6           192.168.0.3           ICMP     Echo (ping) request
61 12.386556   192.168.0.6           192.168.0.3           ICMP     Echo (ping) request
62 12.418465   192.168.0.3           192.168.0.6           ICMP     Echo (ping) reply
```

Figure 5: packet capture on $UML5$ (from ethereal)

```
11:44:01.294387 0:c0:9f:16:27:4 0:c0:9f:16:10:3 0800
                98: 192.168.0.6 > 192.168.0.3: icmp: echo request (DF)
11:44:01.314788 0:c0:9f:16:10:3 0:c0:9f:16:6d:2 0800
                98: 192.168.0.6 > 192.168.0.3: icmp: echo request (DF)
11:44:01.314936 0:c0:9f:16:6d:2 0:c0:9f:16:27:4 0800
                98: 192.168.0.3 > 192.168.0.6: icmp: echo reply
11:44:01.326203 0:c0:9f:16:6d:2 0:c0:9f:16:27:4 0806 42:
                arp who-has 192.168.0.6 tell 192.168.0.3
11:44:01.676545 0:c0:9f:16:27:4 ff:ff:ff:ff:ff:ff 0800
                62: 192.168.0.6.654 > 255.255.255.255.654:  udp 20 (DF) [ttl 1]
```

Figure 6: packet capture on $UML3$ with the Ethernet addresses (from tcpdump)

```
Route Table at uml6
------------------------------------------------------------------------------
        IP        |       Seq    |   Hop Count  |     Next Hop
------------------------------------------------------------------------------
  192.168.0.2               1            1         192.168.0.2         Valid  s
ec/msec: 2/827 0
  192.168.0.5               1            1         192.168.0.5         Valid  s
ec/msec: 2/591 0
  192.168.0.6               1            0         192.168.0.6         Valid  s
ec/msec: 172874790/837 1
------------------------------------------------------------------------------

Route Table at uml3
------------------------------------------------------------------------------
        IP        |       Seq    |   Hop Count  |     Next Hop
------------------------------------------------------------------------------
  192.168.0.7               1            1         192.168.0.7         Valid  E
xpired!
  192.168.0.6               1            1         192.168.0.6         Valid  s
ec/msec: 2/227 0
  192.168.0.5               1            1         192.168.0.5         Valid  s
ec/msec: 2/904 0
  192.168.0.3               1            0         192.168.0.3         Valid  s
ec/msec: 172874821/157 1
------------------------------------------------------------------------------
```

Figure 7: AODV routing table at $UML6$ and $UML3$

```
(1) writew(0x93c,HFA384X_PARAM0_OFF)
(2) writew(0xa,HFA384X_CMD_OFF)
        (3) HFA384X_CMDCODE_ALLOC(0x93c,0x0)
(4) writew(HFA384X_EV_ALLOC,HFA384X_EV_STAT_OFF)
(5) writew(0x10,ALLOCFID_OFF)
        (6) evStat=0x18
(7) writew(0x10,HFA384X_EVACK_OFF)
(8) writew(0x8,HFA384X_EV_STAT_OFF)
(9) writew(0x8,HFA384X_EVACK_OFF)
```

Figure 8: Exchange between the hostap driver and the card during the card initialisation phase

```
(1)  writew(0x50,HFA384X_SELECT0_OFF)
(2)  writew(0x0,HFA384X_OFFSET0_OFF)
(3)  netbus_send a=HFA384X_DATA0_OFF len=60
(4)  writew(0x3c,HFA384X_OFFSET0_OFF)
(5)  netbus_send a=HFA384X_DATA0_OFF len=6
(6)  netbus_send a=HFA384X_DATA0_OFF len=1054
(7)  readw(0x30)
(8)  readw(0x38)
(9)  writew(0x460,HFA384X_OFFSET0_OFF)
(10) writew(0x50,HFA384X_PARAM0_OFF)
(11) writew(0x0,HFA384X_PARAM1_OFF)
(12) writew(0x10b,HFA384X_CMD_OFF)
        (13) HFA384X_CMDCODE_TRANSMIT(0x50,0x0)
(14) readw(0x38)
(15) Tx packet at 0x5000, len=1120
(16) readw(0x60)
(17) writew(0x8,0x60)
(18) writew(0x50,0x44)
(19) Interrupt evStat=0x8,inten=0xe09f
(20) writew(0x50,0x14)
        (21) evStat=0x18
(22) writew(0x10,HFA384X_EVACK_OFF)
(23) writew(0x8,0x60)
```

Figure 9: Exchange between the hostap driver and the card during the transmission of a data frame

mand corresponds to the extra feature mentioned previously that is used to transmit an entire buffer in one operation. Line (3) corresponds to the transmission of a frame descriptor and line (6) to the packet in itself. Once all the data have been transmitted, the transmit command itself is finally issued by the driver (line (10) to (13)) with the frame ID as parameter. Then, the card actually transmit the packet in the air (15) and the event status register bits are cleared. To our point of view, this kind of traces might be interesting to understand not only the static behaviour of a chipset as described in the datasheet but also the dynamics of the interactions between the driver and the card.

5 Related work

Several types of emulators have been proposed in the literature. In [10], Keshav et al. virtualised the networking stack of a FreeBSD kernel and allowed to run routing protocols and other networking applications in an emulated environment. Another similar approach is IMUNES proposed in [11]. IMUNES allows a FreeBSD kernel to maintain several networking stacks that are used to support different applications. However, those two solutions do not support wireless interfaces. As mentioned in the introduction, VNUML can be used to simulate wireless adhoc networks. However, this approach is less flexible as it requires an explicit description of the topology. Furthermore, communication links are bidirectional and symmetrical. In contrast, our physical model allows for a more realistic description of the transmission medium and the topology can be modified in real time by a simple "click-and-drag" operation.

Some debugging techniques related to Section 4 are described in [9], chap. 4 and include the use of a kernel debugger or of the Linux Trace Toolkit. Their also exists some I/O analysers that could be used to obtain some traces similar to those shown in Section 4. All these techniques could off course be used in conjunction with the tool presented in this paper. In particular, GDB can be used directly with UML and with the inserted modules. As the software card is "virtually plugged" into the kernel by inserting a module, the UML side of the card may easily be debugged and its interactions with the kernel may be studied with conventional tools.

Other emulation based solutions exist such as for instance *vmware*, *qemu* or *bochs*. While *vmware* does not allow for the addition of new emulated hardware components, *qemu* and *bochs*, which are free software could be used to accomplish the tasks described in this paper. For instance *bochs* supports an NE2000 compatible network card and a wireless interface could be added just as well. However, as stated in the bochs FAQ, bochs emulates every x86 instructions and all the devices in a PC system, it does not reach high emulation speeds.

6 Conclusion and future work

A virtual bus was proposed for User Mode Linux and used to insert the Linux *hostap* driver with nearly no modifications. By connecting multiple UML machines through a physical layer emulator and by providing a GUI for network visualisation, it was shown that this system could represent an interesting approach for wireless network emulation. The utility

of such a method has been illustrated with two practical examples : the testing of an implementation of the AODV routing protocol in a highly realistic environment and the study of the interactions between the driver and the card it drives. Future work will now be focused on

- a complete implementation of the UML PCI interface to allow native drivers to run completely unmodified in UML.
- an implementation of the software card as a Linux kernel thread instead of a separate process. This would greatly improve the performance as it would no longer require any blocking operations for the synchronisation of the card and the kernel.
- improving the readability of the debugging facilities of the emulator. In particular, the *writew* operations initiated by the driver and the subsequent *writew* commands executed by the emulator itself should be reported separately. A GUI could be used to monitor the different status registers in real time.

References

[1] J. Dike. User mode linux. In *5th Annual Linux Showcase & conf.*, Oakland CA, 2001.

[2] D. Fernandez, Tomas de Miguel, and F. Galan. Study and emulation of ipv6 internet-exchange-based addressing models. *IEEE Communication Magazine*, pages 105–112, January 2004.

[3] V. Guffens. A wifi layer for user mode linux. http://www.auto.ucl.ac.be/uml-wifi/.

[4] V. Guffens, G. Bastin, and O. Bonaventure. An emulation infrastructure for multi-hop wireless communication networks. *Internal report*, 2004.

[5] Luke Klein-Berndt. Kernel aodv,national institue of standards and technology , http://w3.antd.nist.gov/wctg/aodv_kernel/.

[6] Jouni Malinen. Host ap driver for intersil prism2/2.5/3,http://hostap.epitest.fi/.

[7] User-mode-linux testing guide - openswan wiki. http://wiki.openswan.org/index.php/UMLTesting.

[8] Réseau citoyen. http://www.reseaucitoyen.be/.

[9] A. Rubini and J. Corbet. *Linux Device Drivers, 2^{nd} Ed.* O'Reilly, 2001.

[10] X.W.Huang, R. Sharma, and S. Keshav. The entrapid protocol development environment. In *Proc. of Infocom*, March 1999.

[11] Marko Zec. Implementing a clonable network stack in the freebsd kernel,. In *Proceedings of the 2003. USENIX Annual Technical Conference, San Antonio, Texas, June 2003.*