PebNet

Michael Sullivan

October 26, 2009
Outline

Introduction

Kernel Extensions

Stack Design
Brilliant Idea: for a Networks class, actually write parts of a networking stack!

But on what platform?
Introduction

Pebbles

- Should need no introduction
- The kernel spec you had to implement in p3
- Small and fairly simple
Introduction

Pebbles
Introduction

Pebbles

- It’s obviously a toy
- But it’s a fairly good toy
- Could be a good base to build something like this on
Kernel Extensions

Problems
Fixes
Ethernet Driver
Thoth IPC
Status
Problems

*Missing features*

- No existing network drivers, etc.
- Not going to ask Networks students to write an ethernet driver...
Problems
Where to put the network stack?

• In Unixes, the network stack traditionally goes in the kernel
• But we certainly don’t want to hand out source for a Pebbles...
• And putting it in the kernel means it can crash the system, is harder to debug, etc
Fixes

* So put the ethernet card driver in the kernel
Fixes

• So put the ethernet card driver in the kernel
• And the rest of the network stack in userland...
Fixes

- So put the ethernet card driver in the kernel
- And the rest of the network stack in userland...
- Communicating via IPC
Ethernet Driver

Kernel Interface

- `eth_send()` - send a frame - fail if no buffer space
- `eth_recv()` - block until a frame is available
- `eth_getaddr()` - fetch the MAC address of a card; kind of hacky and special purpose; could get replaced with a more general `eth_config()`
- These functions all take a card number as a parameter so that multiple cards can be supported
Ethernet Driver
What card to support

We want:

- Simulated by Simics
Ethernet Driver
What card to support

We want:

• Simulated by Simics
• Commonly available in hardware
**Ethernet Driver**

*What card to support*

We want:

- Simulated by Simics
- Commonly available in hardware
- Don’t tell Prof. Eckhardt, but...
Ethernet Driver
What card to support

We want:

- Simulated by Simics
- Commonly available in hardware
- Don’t tell Prof. Eckhardt, but... simulated by QEMU
Ethernet Driver
Card choice

The Intel 8255x “Fast Ethernet” family of cards.
• Simics simulates the 82559
Ethernet Driver

Card choice

The Intel 8255x “Fast Ethernet” family of cards.

- Simics simulates the 82559
- QEMU simulates the 82557b and the 82559er
**Ethernet Driver**

*Card choice*

The Intel 8255x “Fast Ethernet” family of cards.

- Simics simulates the 82559
- QEMU simulates the 82557b and the 82559er
- Fairly well documented
Ethernet Driver

Card choice

The Intel 8255x “Fast Ethernet” family of cards.

- Simics simulates the 82559
- QEMU simulates the 82557b and the 82559er
- Fairly well documented
- Not *that* much of a CF
Ethernet Driver
Also readily available...
*Ethernet Driver*

*Card Initialization*

- Probe PCI bus looking for card we can handle
Ethernet Driver

Card Initialization

- Probe PCI bus looking for card we can handle
- Reset the card, turn on the card, enable PCI bus access
Ethernet Driver
Card Initialization

- Probe PCI bus looking for card we can handle
- Reset the card, turn on the card, enable PCI bus access
- Read the MAC address out of the card’s onboard EEPROM
Ethernet Driver
Card Initialization

- Probe PCI bus looking for card we can handle
- Reset the card, turn on the card, enable PCI bus access
- Read the MAC address out of the card’s onboard EEPROM
- Configure card settings (24-byte array of settings)
**Ethernet Driver**

**Card Initialization**

- Probe PCI bus looking for card we can handle
- Reset the card, turn on the card, enable PCI bus access
- Read the MAC address out of the card’s onboard EEPROM
- Configure card settings (24-byte array of settings)
- The less said about these last two steps the better.
Ethernet Driver
Card Initialization

- Probe PCI bus looking for card we can handle
- Reset the card, turn on the card, enable PCI bus access
- Read the MAC address out of the card’s onboard EEPROM
- Configure card settings (24-byte array of settings)
- The less said about these last two steps the better.
- “And when you gaze long into an abyss the abyss also gazes into you.”
Diagram of receive setup; actually set up as a circular list. The transmit descriptors look basically the same except that they point to a buffer not directly attached to the descriptor.
Ethernet Driver

Driver Operation

- `eth_send()` fills the first available buffer, restarts the transmit unit
- `eth_recv()` blocks until there is a full buffer, restarts the receive unit if it had stalled
Some sort of IPC mechanism is needed to tie together a userspace based network stack
Some sort of IPC mechanism is needed to tie together a userspace based network stack

But what kind?
Some sort of IPC mechanism is needed to tie together a userspace based network stack

But what kind?

“The nice thing about standards is...”
Thoth IPC
The model

- Thoth-like port based IPC (s08’s p4)
- Communication is done through “ports” which are owned by tasks
- send() sends a message from a source port you own to a destination port, and then blocks until another thread has done a receive() followed by a reply()
- receive() waits for a message to be sent to a port, and informs the caller who sent the message
- reply() sends a reply to a message that was previously received()
**Thoth IPC**

**Advantages**

- Pretty simple
- Fits well with a “remote procedure call” model where one process requests a service from another and waits for a response
Thoth IPC

Disadvantages

- Nontrivial uses of ports essentially require the application to be multithreaded
- Lots of ways to fall into a hole: if they other guy never reply()s, you are hosed.
- Flat port namespace
Status

- Platform for work: YAPI (Yet Another Pebbles Implementation) by Michael Sullivan (mjsulliv@) and Anand Subramanian (asubrama@).
- 8255x network driver and Thoth IPC implemented and working.
**Status**

```
<table>
<thead>
<tr>
<th>Trying to receive something?</th>
</tr>
</thead>
<tbody>
<tr>
<td>[net] (3): waiting for a packet...</td>
</tr>
<tr>
<td>[net] (2): got a packet of size 60</td>
</tr>
<tr>
<td>got 60 bytes</td>
</tr>
</tbody>
</table>

---

```
Stack Design

Design
Demultiplexer
Design

Overall design

- Put each layer in the stack in its own process, communicating with the others through IPC.
- This means separate processes for Ethernet, ARP, IP, UDP, TCP...
Design Diagram
Design
Advantages

• Fairly clean and modular.
Design
Advantages

- Fairly clean and modular.
- Easy to give out binaries for $N - 1$ modules and have a student write the remaining one.
Design

Disadvantages

- The protocol stacks have layering violations, so the boundaries are something of a lie. Specifically, the UDP and TCP checksums include the IP header, so the IP daemon will probably need to know how to do checksumming for TCP and UDP.
Design

Disadvantages

- The protocol stacks have layering violations, so the boundaries are something of a lie. Specifically, the UDP and TCP checksums include the IP header, so the IP daemon will probably need to know how to do checksumming for TCP and UDP.
- DHCP is technically over UDP/IP, but is very special and violates a lot of the normal rules. It will talk directly to the Ethernet daemon...
Design

Disadvantages

- The protocol stacks have layering violations, so the boundaries are something of a lie. Specifically, the UDP and TCP checksums include the IP header, so the IP daemon will probably need to know how to do checksumming for TCP and UDP.
- DHCP is technically over UDP/IP, but is very special and violates a lot of the normal rules. It will talk directly to the Ethernet daemon...
- “Suffers from the microkernel.” *Lots* of context switches and copies required to send/receive packets from a client application. The copies *could* be mitigated through zero-copy IPC and network calls, but the context switches are harder to avoid.
Demultiplexer

- One very common pattern is receiving data buffers from some source, performing some processing, and then demultiplexing it to another client based on some tag in the message.
Demultiplexer

- One very common pattern is receiving data buffers from some source, performing some processing, and then demultiplexing it to another client based on some tag in the message.
- Potentially multiple clients interested in messages with some tag.
• One very common pattern is receiving data buffers from some source, performing some processing, and then demultiplexing it to another client based on some tag in the message.

• Potentially multiple clients interested in messages with some tag.

• This shows up in Ethernet, IP, UDP.

Demultiplexer
Demultiplexer

- One very common pattern is receiving data buffers from some source, performing some processing, and then demultiplexing it to another client based on some tag in the message.
- Potentially multiple clients interested in messages with some tag.
- This shows up in Ethernet, IP, UDP.
- Want a generic facility for demultiplexing messages to ports based on a tag.
Goals

- Extend kernel with ethernet driver, IPC.
- Use DHCP to get an IP address.
- Be able to respond to pings
- Have some reasonable framework for continued work
- Initialize two cards (of the same type)