Barrelfish Networking Architecture

Kaveh Razavi

Departement of Computer Science, ETH Zurich

razavik@student.ethz.ch

Abstract

This distributed systems lab project is about extending Barrelfish networking architecture. Before this project, the system could only support a single application for network interaction but now it can support several. To make this happen, extensive changes in the e1000 NIC driver had to be introduced. A previous lab project named bfdmux had to be integrated within Barrelfish which included a simple virtual machine for demultiplexing incoming packets to different users and doing sanity checks on outgoing packets. Finally, a networking daemon is now necessary for client management, namely port allocation and security checks.

1 Introduction

Barrelfish is a research operating system being developed collaboratively by researchers in the Systems Group at ETH Zurich in Switzerland and in the Systems and Networking Group at Microsoft Research Cambridge in the UK. It is intended as a vehicle for exploring ideas about the structure of operating systems for hardware of the future. As networking is an important part of the current operating systems, it is necessary for a research operating system like Barrelfish to provide equal support for networking so that it becomes possible to measure its efficiency compared to other operating systems when dealing with high throughput networks. Before this project, the webserver of the Barrelfish was providing memory directly to the NIC driver. This is an unlikely setup in the current operating systems because it makes it impossible to support several networking applications. The purpose of this work is to extend this architecture as much as possible with minimal changing of the current interfaces which defines the interactions with the networking stack (LWIP).

To enrich the networking support, the NIC driver was extended to use a private buffer for incoming network packets. Then a virtual machine, which is executing inside the driver context, iterates over the filters provided by the netd (networking daemon) to find a match to a user application. These filters are generated using the local port and the local IP address of which the network application is using to communicate with the network. If a match is found, the incoming data is copied to the user provided memory and a message saying a packet is received is submitted to the user application by the driver. On the transmit side, to enforce validity, another sets of filters are generated to check the outgoing packets. This is done by copying the header of the packets into a ring of private memories and then running filters on them and transmitting them, just in case one of the filters succeed. This is a rough scheme of how things are looking now and I am going to explore them in detail later. In other words, I explain the process of refining the networking architecture of Barrelfish from figure one to figure two.

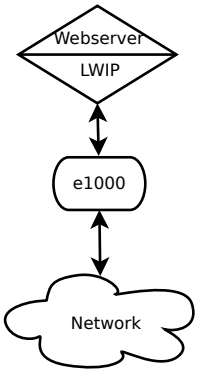

Fig 1. Barrelfish netwoking architecture before my project

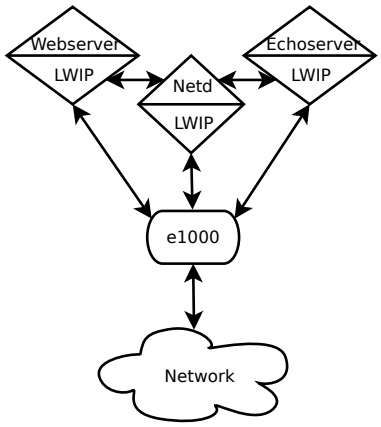

Fig 2. Barrelfish netwoking architecture after my project

Next chapter is dedicated to the Barrelfish demultiplexer, the VM and the filter codes. Chapter three is about the netd, filter generation and port allocation, and how it enforces integrity and security. Chapter four

describes how memory is managed inside LWIP before discussing the interface files and functions required by a driver to work in Barrelfish. In chapter five, the current state of the new e1000 driver is explained while chapter six explains the changes which needed to be introduced in the LWIP level. The last chapter discusses possible future work on the Barrelfish networking architecture.

2 Barrelfish demultiplexer

This section is mostly about the implemented Barrelfish Demultiplexer and how it is now integrated inside Barrelfish. If you want to write your own NIC driver, you would probably like to read this section carefully. It consists of two parts: the virtual machine and the filter creation and compilation. These are developed by Amin Baumeler and Rainer Voigt during the course of their distributed systems lab project.

2.1 The virtual machine

The virtual machine is a light weight code which takes a filter byte code and a packet and then figures out whether the filter matches the packet or not. Internally it is a recursive call over a switch code which checks the filter opcodes against the packet data. I had to integrate the virtual machine code inside the Barrelfish and now it is compiled as a Barrelfish library called bfdmuxvm. To be able to use the VM you need to add this library to the Hake file and then include the appropriate header. The sample usage follows:

```
1 res = execute_filter(filter, filter_len, packet, packet_len, &error);
2 if(res) {
3 // succeed
4 }
5 else {
6 // failed
7 }
```

2.2 The filter generation and compilation

As a network application, you need to have your filters registered by netd to the driver. Two steps are required to do so and they are done by netd on behalf of the network application. For generating a filter, you can use another library which is called bfdmuxtools and it contains a rich API for filter generation based on the source and destination MAC addresses, IP addresses and ports. I needed to change a lot in there and enrich its API more since it was working on the network layer but filtering is now happening on the link layer. Thus, a further set of API to create ARP filters and also IP filters with MAC with the same fashion was included in the original code. I also needed to port a libc posix function from the FreeBSD source tree to make the library compile inside Barrelfish and also some cleaning up to resolve some conflicts in naming with LWIP.

The API could be found in include/bfdmuxtools/tools.h. For generating filters using the API and compiling them, one could use the following code:

```
1 // This builds a filter for incoming packets with any source IP and PORT,
2 // which have the given MAC address, IP address and port as their destination.
3 // One can see that this filter is suitable when an application intends to open a port.
4 filter = build_ether_dst_ipv4_tcp_filter(mac, htonl(local_ip.addr),
5 BFDMUX_IP_ADDR_ANY, (port_t) port, PORT_ANY);
6
7 // This compiles the filter and generates the bytecode to be executed by the VM
8 compile_filter(filter, &filter_bytecode, &filter_bytecode_len);
```

3 A networking daemon for Barrelfish

What we call a networking daemon is a trusted application, developed during this lab project, running on a separate domain, which provides a service for port allocation/deallocation to the network applications and also implements functions to register/deregister filters on the driver.

There are numbers of questions that need to be answered before designing and using a networking daemon for Barrelfish. From the previous definition of networking daemon, one might ask why we are decoupling these certain tasks from the driver itself. From my point of view, there are three good things about having a completely different domain for the networking daemon. First and most importantly, it is a lot cleaner because it completely decouples packet management from packet transfer. Secondly, not linking another library (given we make networking daemon as a library) to the driver, makes it smaller and thus more cache friendly. The second argument might not be strong due to the increasing local core cache size and usually the actual driver code itself is not that bulky. The overhead of having networking daemon on another domain is the added latency due to URPC between the driver and the networking daemon domain which is not in the fast path though. One should also note that making the networking daemon as a library does not allow for central management.

Measuring the performance of these two approaches under different workloads is out of the scope of this lab project but as mentioned, the current design is cleaner and goes nicer with the microkernel properties of a multikernel. The last good thing about making netd like this is about the question of whether we should have a networking daemon per driver or should the networking daemon handle all the drivers in the system. The current design is very flexible and can easily be extended to support both approaches as we only have one driver in the system now.

Apart from providing management service to clients, netd does a couple of other tasks as well. It enforces integrity and security by maintaining a list of open and closed ports in the system and per user as well. The details of this will be discussed soon. Another tasks include responding to ARP requests, doing DHCP at startup to get an IP and handling packets which do not match any other filter in the driver.

3.1 Port allocator/deallocator

As a part of the new design, port allocation and deallocation needs to be explicitly done in the netd instead of LWIP. I used the port allocator provided by Adrian and modified it a little bit to work with the netd. It provides routines for opening a specific port of a given type (UDP/TCP) or a non-specific one. This allocator is then later used to provide part of the netd service. Here is the API provided by the port allocator:

```
/*****************************************************************************
 * Prototypes
 *****************************************************************************/

void init_free_ports(void);
uint16_t alloc_tcp_port(void);
uint16_t alloc_udp_port(void);
uint16_t alloc_specific_port(uint16_t port, uint64_t type);
void free_port(uint16_t port, uint64_t type);
```

3.2 The interface netd.if

The service provided by netd is defined in the interface definition file netd.if. First of all you can query the network for the current IP address of the system. This is required to create packets at LWIP level. The other functions include request for opening an arbitrary UDP/TCP port (get_port) or an specific one (bind_port). It does provide a function for closing a UDP/TCP port as well.

The back end for this service is in idc_barrelfish.{c, h} in the netd. Basically what happens during opening a port is calling one of the port alloc's API depending on the RPC call (get_port or bind_port). If there is no error, then netd proceeds with creating two filters, one for receive and one for transmit and then compiling and registering them. After they are registered inside the driver, netd sends a message saying the requested port is registered or something went wrong in the middle.

During a close port call, netd deregisters the filters in the driver and then deallocates the closing port, making it available to other clients. Also, when an URPC connection to a client is lost, netd assumes the client has died and handles this situation gracefully by deregistering all of its open ports' filters in the driver and then it deallocates all of its ports and frees all of the relevant structures.

3.3 Enforcing integrity and security

To enforce integrity and security, netd uses the following data structures defined in netd.h:

```
/**
 * This is the structure used to hold all of the ports allocated to a
 * networking application and their relavanet buffer. This also keeps the
 * state of the filter registration/deregistration sequence.
 */
struct buffer_port_translation {
    uint64_t buffer_id_rx;
    uint64_t buffer_id_tx;
    uint64_t filter_id;
    uint64_t type;
    uint16_t port;
    bool active;
    bool bind;
    bool closing;
    struct netd_service_response* st;
    struct buffer_port_translation* next;
};
```

```
18
19 /**
20  * Represents a network user application. This can easily be extended later.
21  */
22 struct net_user {
23     struct buffer_port_translation* open_ports;
24     bool died;
25     struct net_user* next;
26 };
```

Keeping the state of users and their ports in these structures make things easy, for example deregistering/deallocating ports upon exiting or checking invalid requests for closing or opening a port on different buffer IDs. The relevant error codes for such situations are all defined in errno.fugo file under NETD and are handled gracefully and propagated to the relevant client.

4 Developing a NIC driver for Barrelfish

Every NIC driver for Barrelfish should export itself as two services. One should be compliant with the interface definition ether.if, which defines the communication channel between LWIP and the driver. The other is defined in ether_netd.if, which defines the communication channel with the networking daemon.

This section will continue talking about what is supposed to be done in the message handlers of the interface definitions, in order to make the device driver work properly. How to design the handler functions and their underneath functionality is the device driver developer's task and might very well be dependent on the actual NIC hardware.

Before we move on to the interface files, we need to discuss how LWIP communicates with the device driver for receiving and transmitting network packets.

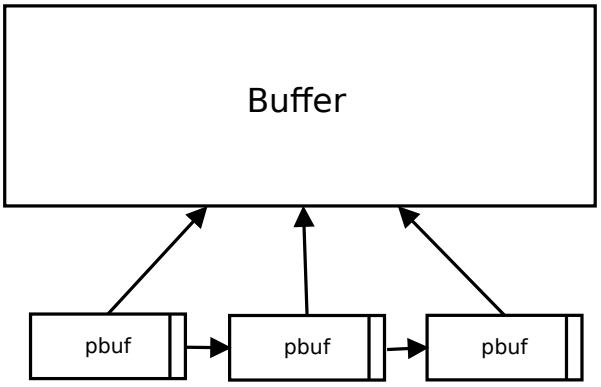

Fig 3. pbufs and their designated buffer

4.1 LWIP Communication concepts

The basic block for transferring packets in LWIP is pbuf. It also is the basic unit of memory management in LWIP. A pbuf is basically a pointer to a piece of memory (payload) plus a pointer to the next pbuf. When sending or receiving packets, packets are constructed (transmit) or reconstructed (receive) by a singly linked list of these pbufs. The pbuf payload is a pointer to a pre-registered chunk of memory called a buffer. Figure three shows what just discussed.

In Barrelfish, these buffers are registered when clients connect to the driver. In the current version, one buffer usually deals with receiving packets and the other one with transmitting them. However, there are exceptions for optimization purposes like reusing receive pbufs for answering to ARP requests.

4.2 The interface definition ether.if

This interface definition file which defines the service provided by the driver for an LWIP instance remained unchanged except one more added part. It lets a client register a chunk of memory for pbuf management inside the driver. Transmit pbufs as before are sent by calling transmit_packet and processed upon a tx_done response. Receive pbufs are registered by calling register_pbuf and when a packet which belongs to the client is received, a packet_received is sent from the driver to the LWIP with the pbuf identifier (it is the LWIP virtual address of the head pbuf structure).

4.3 The interface definition ether_netd.if

The connection channel between the driver and the networking daemon is defined here. At the start, netd should register a chunk of memory for filter data transfer with register_netd_memory.
For registering filters, it can use register_filter. For each call to register_filter, two filters are registered: one for receive and one for transmit. The one for receive will decide whether a packet matches the port, and the one for transmit does sanity checking only. The response by the driver is the filter_id for the registered filters.

When closing ports, netd needs to deregister filters. It does so by calling deregister_filter and providing the filter_id and the receive and transmit buffer ids as well.

Since netd needs to know what is the MAC address of the card to be able to create valid filters, get_mac_address needs to be there as well. The last function provided is for registering general receive and transmit ARP filters. I will discuss why they need to be there in detail later on.

5 E1000n: a NIC driver for Barrelfish

In this section, the current state of the e1000 driver is discussed in detail. First, we focus on receive and transmit architectures and then we move on to other features of the device driver.

5.1 Receive

At the start of this project, there was only one network application in the system. With that setup, receive pbufs were registered directly into the device receive ring and transmit pbufs were registered in the device transmit ring without any sanity checking.

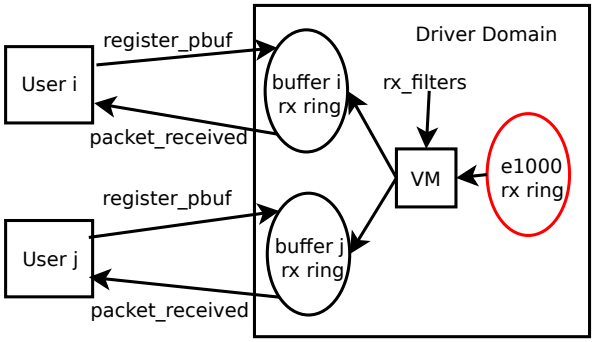

Fig 4. The receive architecture

5.1.1 Receive private memory and user receive ring

The first thing which needed to be changed when I started doing this project was changing the receive mechanism since it was not possible to register pbufs from LWIP to the device receive ring directly anymore. This was due to the fact that now there is only one receive ring in the e1000 device but there could be more than one networking application. Thus, now the e1000n driver has a private memory for receiving packets. The receive descriptors are registered in the device ring with this private memory upon driver initialization. When a packet is received it is copied to the user or discarded and the private memory is added back to the ring so that the device never runs out of memory.

This also implies that the pbufs registered by clients, should be handled internally by the driver. To do that, we created a receive ring for each client connection (which indicates a buffer), which resembles the actual device ring. We copy packets to this ring in case of a match. Thus, from the lwip point of view, nothing has changed regarding the packet transfer. The memory required for client pbuf management is also provided by the client at start up. Figure 4 presents the implemented architecture for receive.

5.1.2 Receive filters

As you can see in figure four, there are driver wide receive filters which are used by VM to decide which client the packet should be copied to. As discussed before, this list is maintained by netd.

5.2 Transmit

As mentioned before, when a port is opened for packet transfer, two filters are registered for it. We discussed the receive filters and now we are going to discuss the transmit filters.

5.2.1 Transmit filters

In principle, if we trust the user applications not to send any malicious packet, transmit filters are unnecessary. However, in real life there could all kind of malicious applications running intentionally or unintentionally (e.g. a worm which exploited a software vulnerability to get into the system domain). Transmit filters thus check the sanity of packets being sent to the network.

Upon opening a port, a transmit filter is registered with the transmit buffer of the networking application. It checks whether a packet which is being sent from this buffer, has correct source MAC, IP and port. There is also a driver wide generic filter which checks for ARP packets and lets them pass.

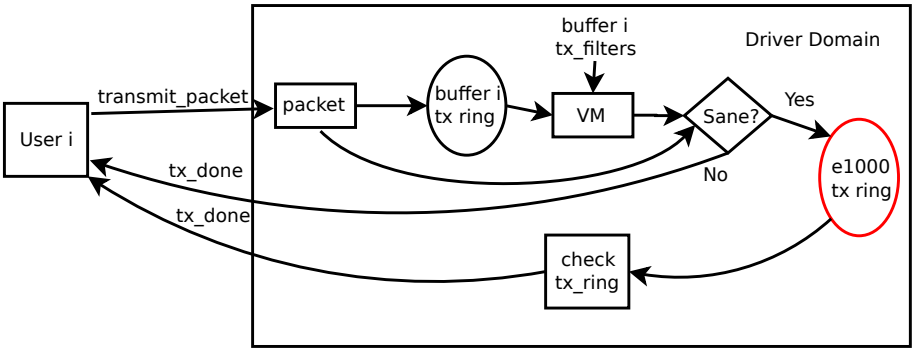

Fig 5. The transmit architecture

5.2.2 Transmit private memory ring in the driver

To be able to do this, it is necessary that the part of the packet that contains the header, is copied into a private memory inside the driver, so that a malicious user application would not be able to change this after the check is performed.

Each buffer now has a ring of private memory chunks with the size of the last byte accessed by the filter. When a packet which consisted of multiple pbufs is enqueued for transmit, the driver goes over them and copies data till enough data for filter execution is copied. Then it executes the filters associated with that buffer. If one succeeded, it enqueues a tx descriptor inside the card transmit ring with the private memory physical address. The remaining pbufs are sent out like before. If no filter succeeded, nothing is sent out to the wire and the user application is notified with pre-mature tx_done to avoid memory leaks on the client side.

A ring is necessary to avoid race when user application is trying to transmit a lot of packets. Figure five shows roughly how things look like. Note that the ring is checked every now and then by a routine and in case a packet is transmitted, a tx_done is sent to the user.

5.3 Dealing with IP fragmentation

Before this project, LWIP handled IP fragmentation. Since we copy packets to the LWIP instances only when a filter succeeds on the TCP or UDP header, we need to find a way to recognize and copy fragmented packets to the right client because not all of them have the header.

Thus, I implemented a basic IP defragmentation algorithm to deal with this. It handles fragmented IP packets by creating filters on the fly when a fragmented IP packet is received. Since it needs to be efficient, I did not use bfdmuxtools and bfdmuxvm. Instead, I did it by checking some certain bytes in the packet data.

The algorithm also deals with out of order fragmented packets. Expiring fragmented packets in case they can not be defragmented due to packet loss and also handling out of order packets which arrive after the packet which marked as the last one are in TODO list. They basically need enabling a timer in the e1000 driver for expiring packets and on the fly filters.

5.4 Handling ARP

ARP packets are very special in the respect that they are working at the IP layer but are needed by LWIP instances nevertheless. One of the ideas of having netd is handling IP layer packets so that the clients would not need to deal with them since working on IP layer packets is usually a privileged task.

However, after implementing the basic system, I observed that for example the webserver does not start copying data from NFS. Digging into it, it became clear that it requires the NFS server MAC address to be able to construct the packet. It was sending out an ARP request for NFS Server MAC address, but the response was being copied to the netd since we believed that netd should deal with it.

To handle this, we decided to replicate ARP responses to all the instances of LWIP since they share the same subnet mask. Thus, at system initialization netd registers a generic receive filter which copies the ARP responses to all network clients. Outgoing ARP responses and requests are also matched against a generic ARP transmit filter which is registered by netd at system initialization as well.

6 Changes introduced in LWIP

This section is about the changes that needed to be introduced in the LWIP so that it could work in the new architecture. The first part is about the changes concerning the new driver interface and the second part is about the changes concerning the netd.

6.1 LWIP and the new ether.if

One would assume that no changes would be necessary since we tried to avoid any changes to ether.if during all stages of development. This is mostly true except the fact that the previously developed LWIP IDC code, assumed only two buffer IDs (zero and one) for the LWIP instance in some places. For example in the old code of mem_barrelfish_get_buffer_desc it was assummed that there are only two buffer IDs and it made some problems later when we loaded more networking applications at startup.

Except this little change to handle different buffer IDs, no more changes needed to be introduced in LWIP.

6.2 LWIP and the networking daemon

First of all, some message handlers needed to be developed as LWIP is now a client of the netd. As mentioned previously, for opening or closing a port, a call needs to be made to netd. Thus, a set of IDC functions to make these calls were also added to the relavant idc files. We needed to handle special cases for netd because it is also linked with LWIP and can not send messages to its own when opening a port. To do this, we pass some function pointers to lwip_init in case of netd and instead of making an IDC call, we just call these port management functions locally. Opening a port is sometimes required in netd; For example, while doing DHCP.

LWIP provides functionality for port management and it was being used in the old code. However with introducing a networking daemon, port management moved there. As a result, I needed to look in all places where LWIP opens or closes a port and comment the port management code and do an idc call instead. One can observe that IDC calls need to be polling this way.

7 Future Work

There are many things left to do in the networking domain of Barrelfish. For example, One important thing is adding the ability to authenticate netd. Currently, the driver assumes the first connection to be from netd. Making the netd driver aware and handle more than one driver is another thing which needs to be added when another NIC driver is developed for Barrelfish. There should not be a lot of work needed for this though. Optimizing the network path even more by handling fragmented packets somewhere else than inside the driver is another thing to do. The most important future work is going to be porting the current changes I have made using old IDC to the new IDC.

References

[1] LWIP STABLE-1.3.1

[2] Barrelfish source code, 2010 February.

[3] Amin Baumeler, Rainer Voigt: bfdmux Barrelfish Demultiplexer, September 2009.