RC22630 (W0211-054) November 6, 2002
Computer Science

IBM Research Report

Multi-Personality Network Interfaces

Eric Van Hensbergen, Freeman L. Rawson
Austin Research Laboratory
International Business Machines
Austin, TX 78758

Multi-Personality Network Interfaces

Eric Van Hensbergen Freeman Rawson

evanhensbergen@us.ibm.com frawson@us.ibm.com

Austin Research Laboratory
International Business Machines
Austin, TX 78758

October 31, 2002

Abstract

This report investigates the use of network interface (NIC) programming to provide multiple personalities or styles for the network interface, including ones that are similar to traditional storage and serial device interfaces. Doing so simplifies the implementation of other types of I/O access over the network at the link layer and helps advance the notion that the network can become the "I/O bus" of clustered systems such as blade servers. This type of support is unnecessary for traditional network environments in which all network access is done through the TCP/IP protocol stack. However, when using link-layer protocols and working directly with commodity network hardware, the mismatch between some types of I/O traffic and the behavior of the network interface becomes apparent, and it is then that the notion of alternative interface personalities becomes valuable. Although there are emerging interconnection architectures such as InfiniBand that offer solutions to this problem, they are currently beyond the range of commodity hardware. In addition, a previous, related effort on link-layer access to remote storage devices, the Ethernet Block Device (EBD), motivates this research. In the course of prototyping EBD, some of the obvious, but often overlooked, differences between network I/O and storage I/O became very apparent, and this effort attempts to solve these problems by concurrently offering multiple interface styles for the network adapter, including one that is better suited for block I/O operations.

1 Introduction

Some recent server blade designs [2], in the interest of minimizing the amount of hardware and the overall complexity of the system design, have had just one input/output connection, the Ethernet interface. This means that the blades are dependent on other server systems for all of their storage resources as well as console access and any other I/O services required. By adding additional hardware and software to certain blades, the system designer can specialize them, creating blades that offer a particular type of I/O service such as storage or console. The resulting hardware, its operating system and the clustering software increasingly treat the network connection as though it were an I/O bus.

Unfortunately, there are some important differences between the way in which traditional network interface hardware operates and the interfaces exported over the I/O bus by storage adapters and consoles. For example, work on the Ethernet Block Device (EBD) [13], which is a link-layer remote storage protocol, highlighted the tendency of an aggressive block device driver to flood the network transmitter when under heavy I/O loads. The flooding is exacerbated by the requirement that the driver decluster write operations, turning one logical output operation created by the operating system to optimize disk operation, into a much larger number of separate network transmissions. Reads do not suffer from this problem since only the request is transmitted, and a single read receives a number of responses that is equal to the total number of blocks that it requests. The relatively small default size of the Linux network transmission queue makes the flooding problem even worse. In addition, while the network driver must provide buffers in advance to hold the unexpected and unsolicited arrival of packets, the natural buffering scheme for storage drivers is to provide all buffers only when the driver makes a request for either input or output. Disk input does not show up unannounced. Changing the personality of the network interface to be more like a storage interface allows the EBD implementation, for example, to send all requests to the device in clustered form and to use the block buffers that it already has rather than allocating and using a separate set of network buffers.

On a blade server with the network interface as the only I/O device, another critical device that it must emulate using the network interface is the console. Although the standard keyboard/video/mouse (KVM) is complex, it is also unnecessary since operating systems and firmware generally support flowing all console traffic over a serial interface. Thus, the problem becomes one of providing enough of a serial interface over the Ethernet network that the serial console code simply works. But the need for console access by very low-level and primitive code such as firmware and early operating system initialization leads to two important requirements. First, the Ethernet interface must offer a serial device emulation that is both detailed, in the sense that the primitives provided are those of the serial hardware, and faithful, in the sense that the apparent behavior is close to that of the serial device. Second, the network interface must do the serial device emulation with essentially no assistance from the host system or the host system's firmware.

Since modern network interfaces, especially ones for gigabit Ethernets, are often programmable and have one or more relatively fast processors and a significant amount of memory, it is now possible to program them to offer more than one personality or style of interface. This research explores the architecture and design of network interface programming that offers multiple interface styles or *personalities* that operate concurrently over the same physical interface. One may also consider this as a form of hardware/software architectural *morphing* or configuration to fit the intended use of the network interface.

This report considers the architecture of the network interface software required to support this environment as well as two important, specific personalities – the traditional Ethernet interface and a storage interface. Additionally, it discusses serial device emulation briefly. The report defines a simple strategy for prototyping and evaluating this idea as well as discussing related and possible future work.

2 Architecture

To architect a multiple-personality network interface, one must understand, in general terms, the types of personalities required. Additionally, there are a number of common details that one must specify including

- the programming environment on the network interface device

- the manner in which the network interface presents the personalities to the host operating system
- the way in which the personalities share the network interface hardware
- the mechanism for error reporting and management
- the relationship to related features such as VLANs [5] and channel bonding.

2.1 Personality Types

As sketched in Section 1, the motivation for this work is to distinguish among different types of I/O especially traditional network I/O, storage access and console support) that have to be done over the network interface on a server blade whose only I/O connection is the network. A number of these differences are readily apparent from prototyping EBD and writing storage networking code at the link layer. More generally, this work identifies three important categories of personality.

- *unsolicited I/O*
- *solicited I/O*
- *device emulation*

Unsolicited I/O is the traditional form of network I/O where incoming traffic arrives asynchronously and without an explicit request from the device or the host. In addition, unsolicited I/O is assumed to do an inline transmission, leading to a significant limitation on the number of transmission requests that the interface processes in a single burst. Solicited I/O models traditional storage I/O, in which the host and the device initiate all I/O transfers, both incoming and outgoing Moreover, solicited I/O does out-of-line transmission and does not such a strict limit on the number of requests that the host can queue to it. Finally, device emulation provides a low-level interface that so closely resembles some other I/O device that device drivers and firmware written for the device being emulated operate properly using it. The only example of device emulation considered in this report is that of a serial device. The fact that the architecture targets three important personality types is not meant to limit it to just those three although detailed consideration of other, radically different I/O personalities is beyond the scope of the current effort.

2.2 Programming Environment

Supporting multiple network interface personalities requires an environment capable of exporting all of them concurrently and in such a way as to manage how the different interfaces interfere with each other. The problem is not to avoid interference, since that, obviously, is not possible, but rather to manage it in a disciplined way.

Since the network interface exports the multiple personalities, it must run some type of programming environment such as a very simple operating system. Section 4 discusses the current state of such programming environments, but they clearly exist and have for some time. All that the architecture requires is a programming environment on the network interface card that allows the concurrent execution of code for each of the personalities and a set of common services including

- interface and packet classification

- configuration management
- packet scheduling and bandwidth management
- error reporting and management.

For reasons described in Section 2.2.2, the personalities require some form of encapsulation, isolating them from each other.

2.2.1 Interface and Packet Classification

For each concurrently executing personality, the network interface programming offers a separate set of interfaces to and from the host system. In particular, this means that the programming environment separates or *classifies* the requests coming from the host and assigns them to the correct personality based on personality-specific information that is passed with each request. It is as if the network interface were a server running several daemon RPC server processes, each with separate sets of entry points for each personality. Figure 1 illustrates this idea, one that is commonly used in other types of distributed storage implementations.

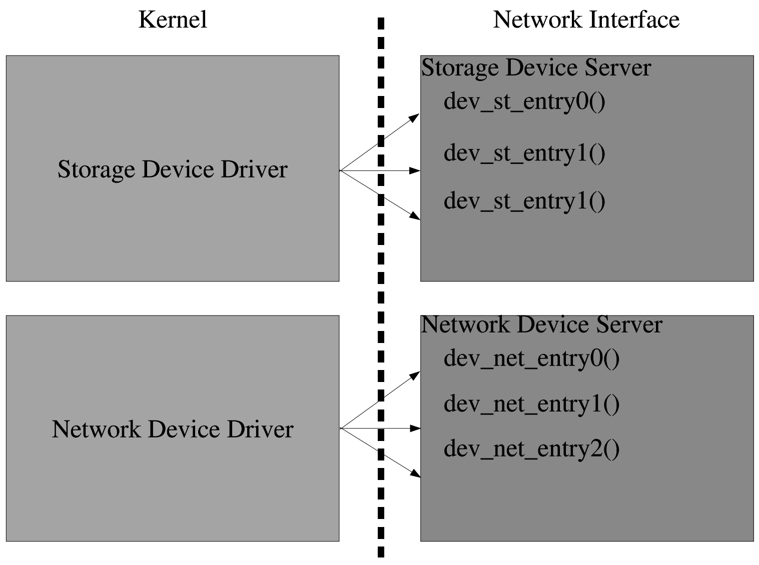

Figure 1: The Network Interface as a Remote Procedure Call Server

In reality of course, there is only one physical interface, and the programming environment contains enough programming logic to assign each host-initiated request to the correct personality. Similarly, the programming environment contains a classifier for frames arriving from the network that allows it, based on the frame type, to assign the frame to a particular personality. Unclassifiable frames on either end are dropped. Figure 2 shows the programming environment's classification scheme.

The frame types used by the different personalities must not overlap to allow the network interface to classify them correctly and route them to the correct personality for presentation to the host. From the perspective of the host operating system, each personality is a separate device with a separate set of apparently independent device resources. This assists in the host-side classification. The programming environment manages the flow of packets through the network interface using a separate flow for each personality. In many ways, this design resembles that of a router with multiple flows, representing different traffic types, across it ([11], [8]). In fact, one interesting way to view a network interface that supports multiple personalities with different types of traffic in an intelligent fashion is as a router between the memory of the host and the network switch. Much of this architecture rests on that analogy. There is another useful analogy between network interfaces with multiple personalities and routers. One can think of each personality as representing a separate logical data plane on the interface while the configuration mechanisms described in Section 2.2.2 constitute the control plane of the interface.

2.2.2 Configuration Management

The term *configuration management* in this context means controlling how many and what personalities the card offers as well as dealing with firmware downloads from the host. The required personality management functions include

- defining what personalities the card offers
- loading the code for a personality
- starting a personality
- stopping a personality
- unloading the code for a stopped personality.

One complicating factor is that many network device drivers assume that they can, at initialization, freely download software onto a network interface card, and their implementations may well depend on the details of the software that they download. This suggests that the best form of programming environment exhibits some hypervisor-like characteristics in that it provides a safe environment for

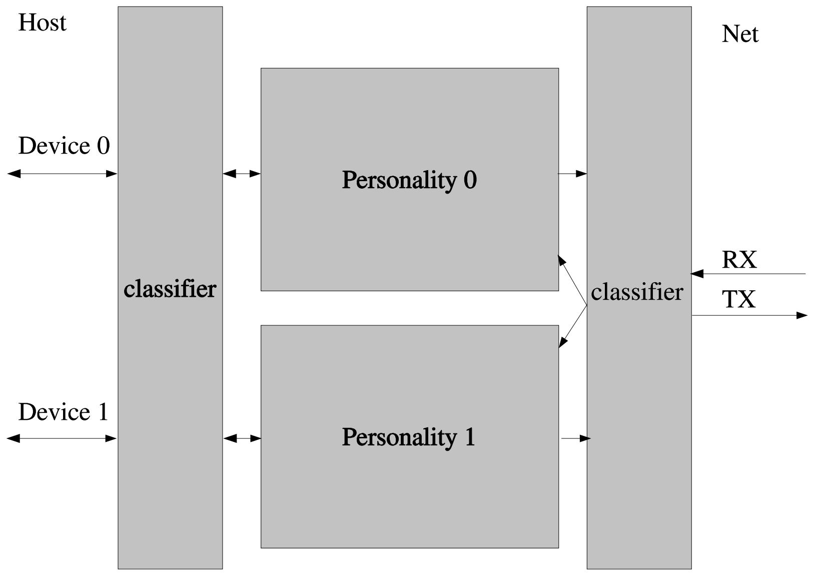

Figure 2: Classification by the Network Interface Programming Environment

the execution of the download without excessive interference with the other, concurrently executing personalities. It is extremely important, for example, that the initialization of the host's network driver not destroy the serial device emulation already set up to service the serial console, but at the same time, it is equally important that the use of multiple interface personalities not require any change to standard, OS-provided, network device drivers.

2.2.3 Resource Allocation and Packet Scheduling

The initial architecture of the multiple-personality network interface does not attempt to specify how the programming environment allocates resources among the personalities. All of the flows are best effort and have limited and fixed queue sizes. Transmission and reception traffic rates which exceed the capacity of the queues and the ability of the network interface to clear them using best effort means cause dropped packets. On the host side, these are reported as queuing errors to the calling program. To reduce the load caused by dropped packets, the packet dropping logic occurs early in the processing sequence. Clearly, there are other packet scheduling approaches worthy of exploration, and Section 5 offers some ideas along these lines.

2.2.4 Error Reporting and Management

Since the personalities share a single set of network interface hardware, except in the channel bonding case considered later, the software environment must report errors to the correct personality based on the operation and frame incurring the error. This means that the programming environment must correctly classify the personality incurring the error and pass the error indication to the personality logic. How the personality handles the error and presents it to the host is personality-dependent. Certain errors are adapter-wide and must be reported through the configuration management subsystem.

2.3 Standard Personalities

To illustrate these concepts, this document describes, from the perspective of the host operating system, the interfaces offered by two of the three personalities that motivate this work.

2.3.1 Traditional Network Interface

The traditional network interface is completely unchanged with the exception that certain Ethernet frame types, those used by the other personalities, are illegal. Currently, only two frames types fall under this restriction, 0x666 and 0x667.

2.3.2 Storage Interface

The overall goal of the storage interface personality is to simplify the code required to implement link-layer network block drivers such as EBD. In particular, this means

- providing solicited I/O for both input and output

- making it possible for the driver to work in terms of storage buffers such as the Linux 2.4 `buffer_head` structures and the Linux 2.5 `bio` structures
- supporting zero-copy input and output by the driver
- offering a simple way to determine how many requests the device can hold in its queue before dropping one
- having a way of notifying a device driver when there is space on the personality's queue
- giving device drivers a simple way to receive responses to their requests.

These features are implemented using a number of interfaces. The interfaces architected here are somewhat Linux-specific although they should move easily to another operating system with different internal representations of block I/O requests and network device structures.

I/O Initiation Using the storage interface, drivers initiate I/O requests with `dev_st_queue_xmit`. The `dev_st_queue_xmit` function which accepts as its arguments a `struct net_st_device*`, a pointer to the storage interface's extension of the standard `struct net_device` structure, and a `struct request*` which points to the block I/O request structure to be queued. This function returns 0 if the request is successfully queued to the device, 1 if the queue is full and other values indicating various types of permanent errors. It is the responsibility of the storage personality to notify the caller by presenting an error response to each buffer passed if there is a transmission error subsequent to the queuing operation. Drivers use this interface to pass both read and write requests to the storage personality.

Queue Management The storage personality presents a set of three queue management primitives that block drivers use to determine the capacity and manage their use of the transmission queue. (It is not entirely clear that the third interface, the upcall registration for space available, is necessary.) The first interface, `dev_st_get_max_queuelen`, retrieves the maximum size of the storage personality's transmission queue while the second, `dev_st_get_current_queuelen`, returns the number of request elements on the queue. Finally, `dev_st_register_queue_callback` registers a function in the block driver that the storage personality invokes when there is available space in the queue: a parameter that is passed to the function indicates the number of available slots. The frames delivered to the storage personality by the standard network device driver drive the invocation of the registered function.

Request Completion Finally, the storage personality must have some way of indicating that a request, or part of a request, is complete. Rather than working on the basis of a whole request, since the operating system may batch unrelated buffers into a single block I/O request, it reports a response on the completion of each buffer of the request, allowing the driver to indicate to the operating system that the buffer is complete. To receive the completion indications, the block device driver uses the `dev_st_register_completion_callback` function, which accepts three arguments, a `net_st_device*`, a two-byte frame type and a pointer to the function that the storage personality calls on receiving an incoming frame of the appropriate type. The prototype of the callback function is `void (*dev_st_completion_callback)(struct net_st_device*, void*)`. When the storage personality receives a frame of the registered type, it invokes the callback function, passing the incoming packet to the driver. The device driver must cast the packet to the appropriate data type.

2.3.3 Serial Console Emulation

The serial console emulation is naturally very device dependent and requires trapping accesses to the I/O addresses (and ports on the x86) that represent the serial device resources as well as presenting interrupts in the same manner and using the same IRQs as the serial hardware.

2.4 VLANs and Channel Bonding

This architecture does not depend on the use of either VLANs [5] or channel bonding. However, the use of VLANs, especially those that frame packets with the IEEE-specified VLAN framing, makes classification much simpler if the personalities map one-to-one to the defined VLANs. This is clearly the preferred configuration. Cisco/Intel-style VLANs that do not use the IEEE-defined VLAN framing have no impact on this architecture.

Channel bonding is ignored by the all personalities in this architecture: they treat the bond device as a standard network device with standard driver interfaces. However, if present, channel bonding may be used in future extensions that do the type of resource management suggested in Section 5.

3 Prototype

Given the complexity of programming the network interface and the limited amount of time to devote to doing a prototype of the multiple personality concept, the implementation proposed below runs entirely on the host system as a Linux 2.4 kernel module. It creates a second set of device interfaces to the network driver and encapsulates the logic that hides the differences between traditional network I/O and storage I/O. Since this section describes only a proposed prototype, it describes only the personalities that the prototype provides and the execution model. The proposed prototype offers the traditional network interface personality and the storage personality. The current design does not include a personality for serial device emulation.

3.1 Storage Personality Data Structures

The data structures that the storage personality prototype maintains provide link-layer storage network drivers to operate solely in terms of the `buffer_head` and `request` structures. The storage personality uses a structure `net_st_device` that describes the personality and its current state. It includes a `list_head` for a chain of `request` structures that are currently outstanding as well as current and maximum count values that simplify the implementation of the queue management functions. It also contains a pointer to the `net_device` structure for the network device itself. The intent of the design of the data structures is to allow the direct use of the `request` structure to manage the transmission of storage network requests. To reduce the number and complexity of retransmission, the maximum number of queued `request` structures is large, generally much larger than the number of network transmissions that the network driver can queue.

3.2 Storage Request Initiation

When the storage network driver calls `dev_st_queue_xmit`, the storage personality determines if there is space on the queue of pending requests. If not, it returns 1 immediately. It then checks the actual network device queue. If there is space for the request, declustered if necessary, it does any declustering required, allocates the required `sk_buff` structures and puts the request or requests on the queue. It also saves a list of the requests queued to the network device along with the addresses of the associated buffers: this list is used to fill the storage network device driver's buffers on reads from the incoming packets. During this time, the code must ensure that there are no other callers putting elements on the queue. Otherwise, if the queue is full, the network code guarantees that `NET_TX_ACTION` will run when the next TX interrupt arrives from the device. If there is not enough space on the queue, the request is placed on the internal list. In either case, the code returns 0 to the caller.

On every `NET_TX_ACTION`, the logic implementing it calls a function provided by the storage device personality that determines if there is space on the device queue for the next request, declustered if necessary, and if so, allocates any `sk_buff` structure required and queues them to the device. In the prototype, the storage personality maintains the driver's requeust ordering although it is by no means clear that this is the optimal design choice.

3.3 Storage Response Delivery

The storage personality acts as a packet handler for frame type 0x667 received on the underlying network device by using the standard packet handling mechanisms provided by the Linux kernel. The kernel invokes the packet handler under the control of the `NET_RX_ACTION` tasklet upon receiption of an RX interrupt from the network interface device. To emulate the behavior of a storage personality running on the network interface hardware, the personality code also copies any incoming data from the standard Linux `sk_buff` structures to the buffer pointed to by the `buffer_head` by matching the incoming packet with an entry on the list of previously sent requests. If all of the expected responses have been received, it retires the request from the list.

3.4 Execution Model

The storage personality executes as in two different environments. On initial entry, when the code is able to queue the decluster request to the real network device, it runs completely under the control of the disk task queue since EBD is a pluggable block driver and does all of its request queue processing when the disk task queue runs. Otherwise, for transmission operations, it runs, as mentioned in Section 3.2, under the control of the `NET_TX_ACTION` tasklet. On the receive side, the storage personality always runs under the control of the `NET_RX_ACTION` tasklet.

3.5 Relationship to Packet Scheduling and Traffic Control

In addition to CPU scheduling, the behavior of the storage personality is also affected by the packet scheduling done by the kernel in its attempt to shape network traffic and do traffic control. This code can change the order in which the system processes packets as well as dropping packets in certain circumstances. As indicated in Section 2.2.3, the current architecture and prototype attempt only best effort, FIFO scheduling on both transmission and reception. Thus, on transmission, the code uses the default FIFO packet scheduler. On reception, packets are passed to the storage personality and the traditional network interface personality in the order received. Clearly, this is an area that merits further investigation as discussed in Section 5.

3.6 EBD Changes

Adding the storage personality requires a number of changes to EBD, most of them involving the removal of existing code that is replaced by functions in the storage personality. In particular, all of the tasklet scheduling code and essentially all of the code dealing with transmitter flooding disappears. In addition, EBD no longer registers as the packet handler for frame type 0x667. Instead, it registers two callbacks, one to handle situations in which space becomes available on the transmission queue and one to receive responses from the server. When processing incoming requests, EBD queues them to the storage personality until the queuing operation returns 1, at which point it turns on flow control. When the storage personality calls the queue space callback, it turns flow control off and resumes queuing to the storage personality for transmission. The buffering inside the storage personality should make this a relatively rare event. The storage interface invokes the second registered callback on the arrival of a response to a previous request. When the callback gets control, any data returned by the server is already in the buffer originally passed on the request. As a result, EBD is able to do "zero-copy" input and output, and there is no need for the code in the current implementation that does zero-copy output using paged `sk_buffs`. In fact, with the storage personality, EBD no longer contains any code for dealing with `sk_buff` structures or directly with the standard network device.

3.7 Evaluation

There are two ways in which one should evaluate this prototype. The first is in terms of the changes made to EBD outlined above. As should be apparent from the previous section, using the storage personality reduces the complexity of the EBD code and allows it to manage only a single type of buffer, the one used by the block I/O subsystem.

The second form of evaluation, one that assesses the performance impact, is more difficult to do since the prototype runs on the CPU with the revised EBD driver. To quantify the performance effect of the storage personality, one may use a scheme that assigns time to EBD and the storage personality. First, one executes a workload using the original EBD driver and determines the amount of time spent inside the driver by instrumenting it with code that records the value of the CPU timestamp counter on driver entry and exit. Then one repeats the experiment with the revised EBD driver. The difference in the time spent inside the driver is an approximate measure of the performance benefit of the storage personality.

3.8 Current Status

The prototype described here has yet to be implemented and evaluated.

4 Related Work

With the advent of network interfaces with powerful on-board processors and relatively large amounts of memory, there have been a number of research and commercial projects that attempt to make use of them to offload the host system or to offer functional enhancements.

4.1 Programming Environments

To take advantage of the newer, more powerful network interface adapters, researchers and developers have written a number of programming environments for use on them. Two early examples of such programming environments are the programming environment for Myrinet developed as part of the Princeton Shrimp project [1] and Spine [3], an extensible operating system for a programmable gigabit Ethernet card. More recently, there is continuing, but as-yet unpublished, work at Rice University on an FPGA-based gigabit Ethernet card that is programmable by the host operating system. Finally, there is a set of emerging commercial products that do TCP/IP and iSCSI [12] off-load, presumably concurrently, so that both standard TCP/IP and iSCSI traffic can share the same physical network interface.

4.2 The Network Interface as an Asymmetric Multiprocessor

Magoutis and others [10] are developing open source implementations of the Direct Access File System (DAFS) [6] on commodity hardware and operating systems. In a related paper ([9]) Magoutis suggests treating the CPU of a commodity system and the processor on the network interface card as an asymmetric multiprocessing system and modifying the interface to the network adapter by splitting the programming logic between the CPU and the network adapter's processor in a non-standard way.

4.3 Uses of Programmable Network Interfaces

There are a number of proposals and projects that present novel uses for programmable network interfaces. A good example is the recent work of Kim, Pai and Rixner [7]. They use part of the memory of the network adapter as a cache to speed the delivery of static web content. As a second example, the purpose of the Shrimp work cited above is to support high-speed user-level communication between machines in a Shrimp cluster.

4.4 More Expressive Device Interfaces

A recent paper by Ganger [4] suggests using firmware programming on disk adapters to support more expressive interfaces than the standard read and write operations between operating systems and storage devices. One can apply the same idea to the storage personality by modifying the personality code that runs on the network interface.

5 Future Work

Beyond the implementation of multiple personalities on network interface hardware, the most important area for further exploration is that of packet scheduling and traffic control. The architecture and prototype discussed above use best effort techniques for handling the traffic associated with each personality and combine them using a first-come, first-served scheme. However, there is no guarantee that this is optimal, and there may even be situations in which this causes incorrect system behavior. Previously, the different types of traffic went over different I/O devices, but with multiple interface personalities and the network as the only I/O device, they are multiplexed onto a single physical connection. This makes it possible, for example, for one type of traffic to starve another. Even if the system behaves correctly, more sophisticated packet scheduling among the personalities may yield performance benefits on important classes of workloads.

Doing sophisticated packet scheduling and traffic management in the context of multiple network interface personalities requires two types of research. First, one needs a way to translate the desired I/O and system characteristics into a set of policies for the traffic flows associated with the individual personalities. Second, the programming environment on the network interface card (or, in the case of the prototype, inside the Linux kernel) must be able to translate these policies into a set of resource allocation and scheduling decisions for the network interface's processor, memory, transmitter and DMA hardware. The paper by Qie et al [11] offers some interesting suggestions along similar lines in the context of network routers.

However, such work offers significant benefits. In particular, it goes even further than the current design does by preventing, rather than merely hiding, the tendency of aggressive link-layer network block drivers like EBD to flood the transmitter hardware. Moreover, with the appropriate policies, a storage personality can incorporate some information about the capacity of the server as well, reducing the complexity of the flow control logic in the device driver.

Another possible area for future investigation is the development of more sophisticated and cooperative interfaces between the operating system and the network interface personalities. As mentioned in Section 4 Ganger [4] recently proposed a research program on cooperative interfaces between operating systems and storage devices that relies on programming disk adapter firmware to pass additional information and operation types between operating systems and storage devices. Similar ideas are applicable to all of the personalities offered by a multi-personality network interface.

6 Conclusions

The work described here provides the basis for prototyping multiple interface styles or personalities on a network adapter. However, until the implementation of the prototype described in Section 3 is complete, it is difficult to judge the value of these ideas. If the prototype shows that is is possible to simplify the implementation of link-layer storage networking drivers and indicates that a complete implementation on the network interface would offer superior performance, the idea merits a more thorough study. If, on the other hand, there is no reduction in code complexity or some indication of a performance loss, there is no reason for further investigation.

References

[1] C. Dubnicki, A. Bilas, K. Li, and J. Philbin. Design and implementation of virtual memory-mapped communication on myrinet. In *Proceedings of the 1997 International Parallel Processing Symposium*, April 1997.

[2] W. Felter, T. Keller, M. Kistler, C. Lefurgy, K. Rajamani, R. Rajamony, F. Rawson, B. Smith, and E. VanHensbergen. On the performance and use of dense servers. Submitted to the IBM Journal of Research and Development, 2002.

[3] M. Fiuczynski, R. Martin, T. Owa, and B. Bershad. Spine: A safe programmable and integrated network environment. In *Eight ACM SIGOPS European Workshop*, September 1998.

[4] G. Ganger. Blurring the line between oses and storage devices. Technical report, Carnegie Mellon University, December 2001.

[5] IEEE. IEEE standards for local and metropolitan area networks: Virtual bridge local area networks, IEEE Standard 802.1Q-1998, 1998.

[6] J. Katcher and S. Kleiman. An introduction to the direct access file system. http://www.dafscollaborative.org, June 2000.

[7] H. Kim, V. Pai, and S. Rixner. Increasing web server throughput with network interface data caching. In *Proceedings of the Tenth International Conference on Architectural Support for Programming Languages and Operating Systems*, October 2002.

[8] Eddie Kohler, Robert Morris, Benjie Chen, John Jannotti, and M. Frans Kaashoek. The click moduler router. *ACM Transactions on Computer Systems*, 18(3), August 2000.

[9] K. Magoutis. Design and implementation of a direct access file system (dafs) kernel server for freebsd. In *Proceedings of the BSDCon 2002 Conference*, 2002.

[10] K. Magoutis, S. Addetia, A. Fedorova, M. Seltzer, J. Chase, A. Gallatin, R. Kisley, R. Wickremesinghe, and E. Gabber. Structure and performance of the direct access file system. In *Proceedings of the 2002 USENIX Annual Technical Conference*, 2002.

[11] Xiaohu Qie, Andy Bavier, Larry Peterson, and Scott Karlin. Scheduling Computations on a Software-Based Router. In *Proceedings of the ACM SIGMETRICS 2001 Conference*, pages 13–24, June 2001.

[12] J. Satran, K. Meth, C. Sapuntzakis, M. Chadalapaka, and E. Zeidner. iscsi. http://www.ieft.org/internet-drafts/draft-ietf-ips-iSCSI-17.txt, September 2002.

[13] E. Van Hensbergen and F. Rawson. Revisiting link-layer storage networking. Technical Report RC22609, IBM Research, October 2002.