UC San Diego
UC San Diego Electronic Theses and Dissertations

Title

The networked device driver architecture : a solution for remote I/O

Permalink

https://escholarship.org/uc/item/2hp395ft

Author

Taylor, Cynthia Bagier

Publication Date

2012

Peer reviewed|Thesis/dissertation

UNIVERSITY OF CALIFORNIA, SAN DIEGO

The Networked Device Driver Architecture: A Solution for Remote I/O

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Cynthia Bagier Taylor

Committee in charge:

Professor Joseph Pasquale, Chair
Professor Sheldon Brown
Professor William Griswold
Professor Ramesh Rao
Professor Amin Vahdat

2012

The Dissertation of Cynthia Bagier Taylor is approved and is acceptable
in quality and form for publication on microfilm and electronically:

_____

_____

_____

_____
Chair

University of California, San Diego

2012

DEDICATION

To Neal S. Reeves.

EPIGRAPH

I'll put a girdle round about the earth
In forty minutes. *Puck, A Midsummer Night's Dream, William Shakespeare*

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

LIST OF LISTINGS

ACKNOWLEDGEMENTS

and job talks and kept me from living in a box car.

My office, 3140, is the best office, filled with the best people.

Lastly, thanks to Sophie Natasha Wigglesworth, for all the biting.

VITA

2002	Bachelor of Arts, Oberlin College, Oberlin, Ohio
2008	Master of Science, University of California San Diego, La Jolla
2012	Doctor of Philosophy, University of California San Diego, La Jolla

PUBLICATIONS

C. Taylor, J. Pasquale, A Remote I/O Solution for the Cloud, *5th International Conference on Cloud Computing*, Honolulu, HI, June 2012.

C. Taylor, J. Pasquale, Improving Video Performance in VNC Under Latency Conditions, *2010 International Symposium on Collaborative Technologies and Systems*, Chicago, IL, May 17, 2010.

C. Taylor, The Proximal Workspace Architecture: A Latency-focused Approach to Supporting Context-Aware Applications, *Doctoral Dissertation Consortium, 2010 International Symposium on Collaborative Technologies and Systems*, Chicago, IL, May 17, 2010

C. Taylor, J. Pasquale, Towards a Proximal Resource-based Architecture to Support Augmented Reality Applications, *Workshop on Cloud-Mobile Convergence for Virtual Reality*, Waltham, MA, March 2010

C. Taylor and J. Pasquale, Improving VNC Performance, *UCSD/CSE Tech. Report CW2009-0943*, May 2009.

J.R. Movellan, F. Tanaka, I. Fasel, C. Taylor, P. Ruvolo, and M. Eckhardt, The RUBI project: a progress report, *ACM/IEEE International conference on Human-robot Interaction*, Arlington, VA, March 2007

ABSTRACT OF THE DISSERTATION

The Networked Device Driver Architecture: A Solution for Remote I/O

by

Cynthia Bagier Taylor

Doctor of Philosophy in Computer Science

University of California, San Diego, 2012

Professor Joseph Pasquale, Chair

With rise of both mobile devices and the cloud, we see users frequently turning to remote servers for both data storage and software services, including running applications. However, once applications are no longer co-located with devices, the traditional device driver architecture no longer facilitates communication between them. Frequently, applications must be rewritten in order to receive data from a remote, rather than local, device.

The *networked device driver* architecture is designed to support input/output devices that are separated by the network from the application(s) to which they are

supplying data. The introduction of the network between the device and application also introduces issues such as high latency, low bandwidth, and jitter. We wish to compensate for these problems by alling for the processing of the data sent between the device and application. We also want to maintain network transparency, so that applications do not need to be modified in order to use remote devices.

The networked device driver is split into two parts, one on each side of the network. At one end is the device and its unmodified device driver, and on the other end is the unmodified application. An I/O stream that is sourced at one end and sinked at the other may be modified by a set of pipelined transformation modules. Each module comes in a pair, one on each side of the network, with one side typically applying some operation and the other side applying a corresponding one, such as encoding and decoding the format of the data or pausing and resuming the sending of messages.

We support network transparency with the pairing of modules, guaranteeing that any modification performed on the data stream will be undone before the message reaches the application. We additionally design our system with the goal of supporting ease of customization/extensibility in support of the vastly different needs of various applications and devices that can benefit from remote I/O. In this work, we explore the necessary trade-offs between ease of development and performance, demonstrating that we can leverage many existing mechanisms without creating a limiting amount of overhead.

Chapter 1

Introduction

In this dissertation, we present a new system architecture, the *networked device driver*, designed to support remote input/output devices. This work is motivated by several current trends in computer science: the ubiquity of light-weight, low-power mobile devices, the rise of cloud computing, and computation and data intensive applications which rely on sensor data.

With the rise of smart phones, we now have tiny computers packed with sensors that we carry around with us everywhere. Typical phones have video and audio input, compasses, GPS and accelerometers, among other sensors. Having such sensor-rich devices that are so well incorporated into user's lives opens up amazing possibilities for ubiquitous computing and machine-learning based applications. Because these devices are constantly with us and capable of collecting rich, multi-faceted data, they open up whole new ways to interact with users based on information collected about both the user and their surrounding environment.

We want these applications to be able to both collect a huge amount of data, and to be able to perform computations over this rich dataset. However, because these new mobile devices require a lightweight, small form-factor, they are much more limited in both storage and computational power than most machines today. One solution to this is to move these applications to more powerful machines, and let them communicate with

mobile I/O devices over the network. Applications may run on servers in the cloud, or on home or work machines belonging to the user.

Even if these mobile devices were as powerful as current servers, there are still many reasons to support remote I/O devices. As we build more powerful machines, we also build more computationally intensive applications which produce more and more data, so there will always be applications that can run in real-time only on the most powerful servers. In addition, many sensors and devices are valuable precisely because of where they are located. Scientific sensors which are measuring aspects of the environments in which they are situated can be much more conveniently worked with when using remote access. Robots, drones, and other machines which quickly change location also require remote access of devices. While offloading computation is our main motivation for this work, there are many other reasons to support remote I/O devices.

Typically, once moved across the network, applications must be rewritten in order to process data coming from the network, rather than directly from the input device. This means that in order to relocate an application, one must have the source code for that application, and be able to redesign it. If the source code is not available, this involves tactics such as automatically rewriting the GUI and manually translating and forwarding user input[10, 12]. Even if the source code is available, it opens up issues of parallel code maintenance. Our goal is for the network to be completely transparent, and the device and application to both behave as though they are on the same machine.

Once the application is moved to a remote machine while the I/O devices remain local, issues of network delay have to be dealt with. Updates will be delayed by the trip over the network, and the rate at which they arrive may be uneven. They may also arrive out of order or corrupted. Because the characteristics of the data flow are changed by being sent over the network, we want our software architecture to provide a way to modify the data in order to account for this and counteract the changes effected by the

network.

One approach to solving this sort of problem has been thin client computing, i.e. moving the application from the local machine to a server and forwarding all user I/O between the local computer and server over the Internet. However, traditional thin clients are not designed to work with the plethora of I/O devices currently available, often limiting their support to the keyboard, mouse, and video card. We want a versatile solution that can easily be extended to support any device.

Our architecture is based on the concept of a *networked device driver*, in which a device driver is split into two halves, one half running on the client with the device, and the other half on the server with the application. Network communications occurs between the two halves, transparent to both the device and application. This means that legacy applications can be migrated across the network, with no modification.

In order to allow processing of the data, our architecture supports customizable *transformation modules* that can be designed to dynamically process the device information appropriately. These transformation modules are designed to run in pairs, with one module on the client applying some transformation (e.g. compression, encryption), and the other on the server, reversing the transformation (e.g. decompression, decryption). This allows us to preserve transparency, while supporting processing of data.

Our architecture promotes ease of customization and extensibility (to support new devices). With this in mind, we designed the system to run primarily at user level, rather than within the operating system kernel. This avoids the security issues that come with allowing arbitrary code within the kernel, and allows someone writing modules for our system to leverage existing mechanisms provided by the operating system, such as blocking I/O calls. Our decision to run at the user level required care to ensure that the implementation was not creating significant performance overhead, especially with large updates (i.e. transferring large amounts of data between the modules in our system),

and many transformation modules (that may require many memory copies per update). Consequently, we provide two different mechanisms for transferring data between the system – one using pipes and one using shared memory – and show that high levels of performance are achievable, even with using standard pipes.

We are also motivated by the rise of cloud computing. With more users turning to the cloud for both data storage and software services, there is a rising need for the cloud to support applications that require remote I/O. The networked device driver architecture is uniquely suited to support cloud applications that require input from remote I/O devices.

Applications in the cloud are often executed in virtual machines, allowing for benefits such as consolidation of resources, application migration and security. However, this means that input devices must also be virtualized, adding additional complexity and overhead to the hypervisor [23]. Since our system virtualizes the device driver by splitting it into two parts, the application half of the driver can run entirely within the virtual machine, without the hypervisor having to virtualize the device.

One of the advantages of the cloud is its support for parallel computation, across many virtual machines. The networked device driver architecture supports modules designed to intelligently multiplex and demultiplex information across these machines. Depending on the needs of the application, messages to can be copied to all machines, sent to alternate machines in a round robin fashion, or each message can be divided up, and the parts sent to different machines. The flexibility of the networked device driver architecture easily combines with the different ways the cloud is used.

In summary, we present a new architecture to support remote I/O. This architecture supports network transparency, while still allowing the data passing through it to be processed in order to compensate for the network. It is easy to customize and extend, due both to be very modular, and being implemented almost entirely at the user-level. We present two different implementations, in order to demonstrate the trade-offs between

ease-of-implementation, and performance.

Chapter 1, in part, is a reprint of the material as it appears in the article "A Remote I/O Solution for the Cloud", by Cynthia Taylor and Joseph Pasquale, which appears in the Proceedings of the 5th International Conference on Cloud Computing, Honolulu, HI, June 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 2

The Problem

Assume a device, such as a webcam, is attached to a computer. This webcam produces frames at 30 frames per second, at 600 by 800 pixels, and uses USB to communicate with the computer. The webcam will produce a frame of video, which will be read in by the USB controller, and then sent to the webcam driver. The driver will then forward the video frame to the application, which will display it. (The display process will in turn involve sending information to the video card driver, the video card, the monitor driver, and the monitor.) This is illustrated in Figure 2.1.

However, once the application is moved across the network, to a new machine, this system breaks down, as shown in Figure 2.2. There is no longer a clear path between the device and the application. Frequently, the solution to this is to write specialty helper applications which read from the device driver on the client, and write to the network, and then either rewrite the application to read directly from the network, or write a new helper application on the server side which translates the data from the network to a form the existing application can understand. For the webcam, we would have to write an application that reads the pixel data for each frame from the webcam driver, and then sends this data over the network. We would then rewrite our application to read from the network driver instead of the webcam driver, and to account for any format changes that occurred when packaging the data from the network. It would be necessary to have the
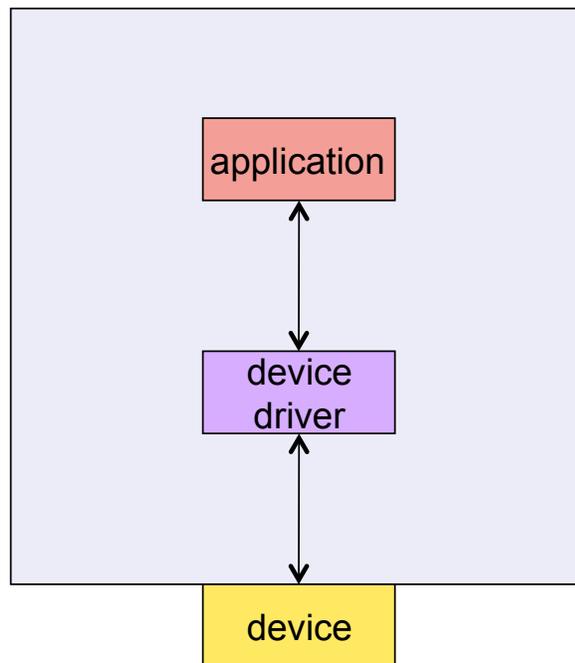
Figure 2.1. When the device and application are co-located, they communicate easily through the device driver.

source code for the application, and then to maintain this new source code, written for the networked device, in parallel with our source code for the local device.

In the networked device driver architecture, the device driver is split into two halves, one of which runs on the client, with the device, and one of which is on the server, with the application. All transmission over the network occurs between the two halves of the driver. This means that the application does not need to be rewritten at all. It interacts with the server half of the networked device driver just as it would interact with the driver for the device. The network is completely transparent to both device and application.

When the device is running on the same machine as the application, several assumptions can be made about how the messages will be delivered from the device to the application. We can assume that messages will be delivered very quickly, that they will arrive in order, and that they will be spaced evenly (i.e. that the time between
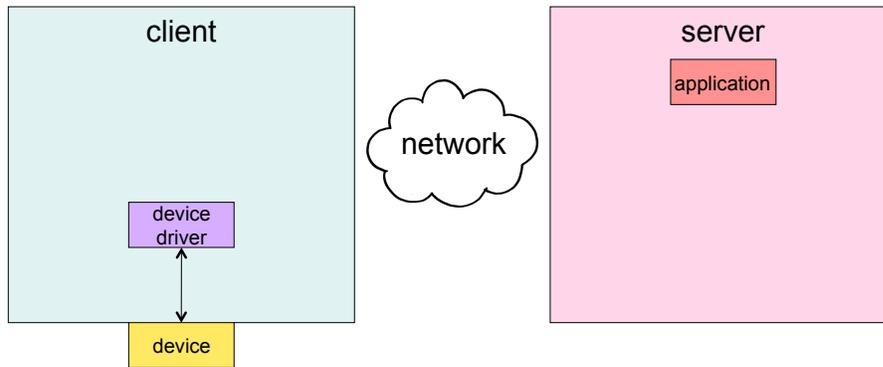
Figure 2.2. When the application is moved to a different machine, there is no longer a clear way for the device and application to communicate.

when two messages are delivered will be very similar to the time between when two messages are received). This is shown in Figure 2.3. We can make these assumptions because data generally travels very quickly within the machine, with minimal external interference. This means that when the application displays frames from the webcam as it receives them, the video will play at the speed it was recorded at, the frames will play at the correct order, and it will be free from jitter, without the application doing any error checking, buffering, or other compensation.

Once the application is moved to a separate machine, we can no longer make these assumptions, as illustrated in Figure 2.4. The network, due to uncontrollable external factors, may add a large delay to the time it takes for messages to reach the application. Messages may suffer from jitter, meaning they will be timed very differently when they arrive for when they were created. And, depending on what network protocol we are using, messages may arrive out of order or not at all. For our webcam and application, this means that the application will need to, at minimum, solve the issues of delay and jitter, usually through buffering. If it uses UDP rather than TCP, messages will arrive more quickly, but there will need to be some way to reassemble messages that arrive out of order, and a way to deal with messages that are lost. This will necessitate rewriting
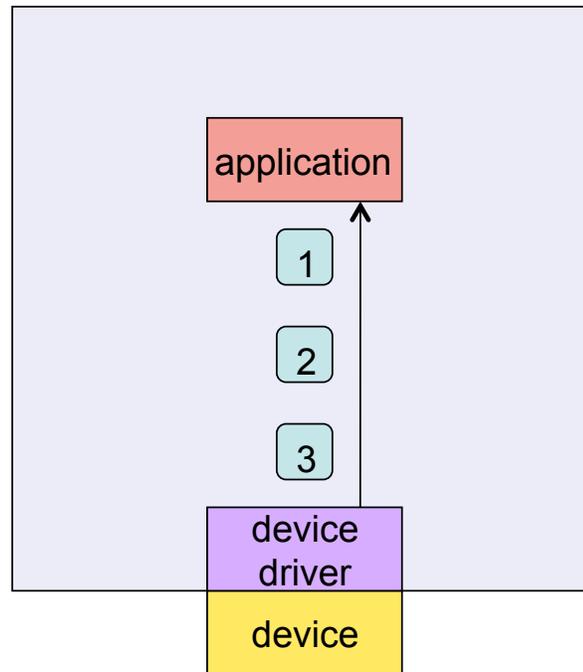
Figure 2.3. When the device and application are on the same machine, updates are received relatively quickly, in order, and with the same temporal spacing as when created.

our application again, in order to add this compensation for the network.

We support customizable modules in our architecture, which allow this processing to occur within the networked device driver, rather than within the application. This means that rather than rewriting the application to add buffering, we can just add a buffering module to the networked device driver. Furthermore, we can add this module to any device that needs buffering, not just the webcam.

In summary, moving applications across the network necessitates finding a new way for them to communicate with devices, as we can no longer simply use the existing device driver. The current solution frequently involves rewriting the application, which is impractical for many applications. We encapsulate the network with a networked device driver, making it transparent to both the device and application. In addition, the presence of the network means that we can no longer assume messages from the device will reach
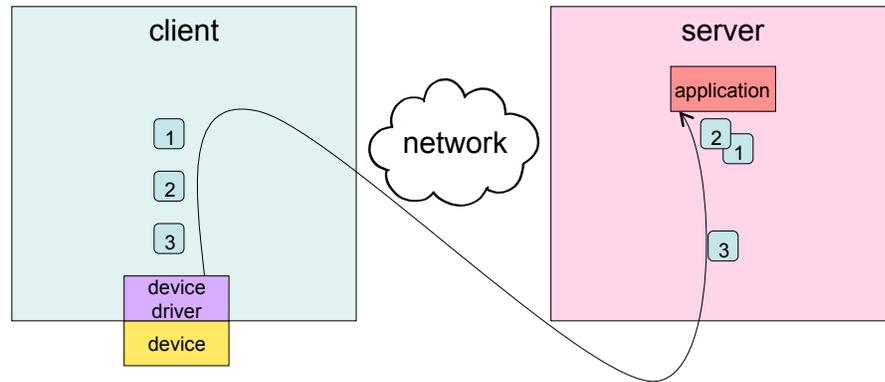
Figure 2.4. When the network is added, updates may be delayed, arrive out-of-order, or not arrive at all.

the application quickly, in order, or at all. Rather than requiring that applications be rewritten to compensate for this, we add support for processing modules to the networked device driver architecture, allowing for processing of data while preserving transparency.

Chapter 3

Related Work

In our related work section, we look at a selection of thin clients and related mechanisms for interacting with devices over the network. We also look at related intra-machine mechanisms. We conclude by describing what our system takes away from the described work, and how it is different.

3.1 X-Windows

One of the classic approaches to sending I/O over the network are demonstrated by X-Windows [20]. X is a window system for Unix that uses a network protocol as its base system. X uses high-level graphics encoding: the server will request the client create a window, text in a specific font, a cursor or an image, and the client will return a handle for it to the server.

Because X is serving as a window manager, using high-level graphical commands, quite a lot of work must be done on the client. The client is responsible for knowing how to create various graphical components and figuring out where to draw them in relation to each other, including overlapping windows. Every time a window is revealed by user movement of components, the client must query the server to find out what to draw in the revealed space. However, the client is sent information from all applications, even ones that are obscured. The client is responsible for figuring out which process the mouse

and keyboard input is meant for, and sending it to the appropriate server and application. The amount of server-client synchronization required for applications and the amount of work the client must do both slow X down. The client workload also means that X is a bad choice for lightweight client devices such as cellphones or PDAs. However, the fact that the client is taking care of multiplexing keyboard and mouse input and generating where application windows appear in relation to each other means that within a user session, different processes can be run on different servers with no apparent difference to the user, a functionality not available in other thin clients. Applications must be written specifically using X in order to be able to use it as a thin client system.

3.2   VNC

The Virtual Network Computing (VNC) [17] System is one of the oldest and simplest thin clients, and defined the first stateless open cross platform standard. The server records changes in its framebuffer, encodes them, and sends them to the client, who updates its framebuffer. All encoding is based around putting a rectangle of pixel data at a given position. The Remote Framebuffer (RFB) Protocol [18] used by VNC defines the following encodings: Raw encoding, where information for each pixel is specified; copy-rectangle encoding, where the client is sent the x,y coordinate to copy the rectangle from, which is useful for scrolling text or moving windows; and RRE, Hextile, and ZRLE encodings, all of which involve setting background pixel data for a large rectangle and then drawing sub-rectangles using either other background colors or raw pixel data. In addition, users can extend the protocol with their own encodings.

VNC uses lazy client-pull updating, where updates are sent only when clients request, and all updates in between client requests are combined into a single update. Connections between client and server are stateless, with no computation being done on the client side other than translating updates and updating the frame buffer. VNC can

be used with unmodified commercial software, on unmodified commercial applications, and can be used to access applications that run on one OS from a computer running a different OS. While it preserves transparency, VNC is limited in that it is designed expressly for mouse, keyboard and video data, and it is difficult at best to add support for new I/O devices.

3.3   USB over IP

Work on USB over IP has focused on sending device information below the driver level by creating a Virtual Host Controller Driver for the USB controller [4]. On the server where the device is physically located, the USB updates are collected by a stub driver below the USB core driver and sent over the network to the client. On the client, updates are sent from a Virtual Host Controller Interface Driver up to various virtualized USB drivers. This has the advantages of both transparency and full functionality. However, there is no way to change the network behavior of different devices or control how different updates are sent.

3.4   THINC

THINC [3] creates a virtual device driver that intercepts drawing commands at the device layer. THINC has a command protocol similar to VNC, with commands RAW, COPY, SFILL, PFILL, and BITMAP. These are also the commands most commonly found in hardware, making it easy for THINC clients to translate from its protocol to the local hardware. THINC captures display commands and translates them into its own commands while preserving any semantic information available about the command. THINC uses a lazy server-push update scheme in which the server stores commands in command queues while waiting to send them to the client. THINC is focused on specific devices (mouse, keyboard, video card, and audio card in THINC), and does not allow for

users to add application-specific modules to process data.

3.5   CameraCast

CameraCast [8] also uses a logical driver designed for video data being sent over a network. CameraCast works as a set of kernel-level abstractions. It extends a standard Linux video interface to allow it to be accessed remotely. It also allows plug ins to process the video, and provides an API for additional functionality for remote video. CameraCast is designed purely for video, and runs entirely at the kernel-level, including user-supplied plug-ins.

3.6   The Berkeley Continuous Media Toolkit

The Berkeley Continuous Media Toolkit creates a development environment for distributed multimedia applications [9]. They create a system architecture where application code is run on top of their services. They provide an extension of the Tcl scripting language that is designed to support multimedia applications with services such as buffering, synchronization, and other abstractions. Their system runs at the application layer. In contrast, our work has the notion of paired processing modules, and does not require applications be written in any specific language or be designed for our system.

3.7   Cascades

Cascades is a system architecture for sensor networks with middleware modules [5]. They use the Python scripting language to create a series of "filters" that can be connected together to modify streams of data from sensors. Our work differs in that it is more general (for more general networks and types of devices), and we use pairs of processing modules.

3.8   Virtual Reality Applications

There is also work in the area of support for virtual reality applications, regarding creating networked VR devices [16, 22, 6]. The Open Tracker work creates a general framework for applications using remote I/O, that includes the notion of "filter" nodes that can modify data [16]. Multi-modal event streams are an extension of Open Tracker that adds abstraction and an event-based message system in order to combine related data from different devices [6]. Both of these systems create generic virtual devices, obscuring the original device, and applications must be modified to communicate with the system, rather than the original device.

3.9   Streams

Other systems have been designed with the idea of being able to modify I/O data. The classic work is the UNIX Streams system [19]. In UNIX Streams, a duplexed I/O stream has two end-points (on the same machine) and intermediate modules which can alter the information being passed through them, allowing users to create their own modules to modify data. This system was designed to function within one computer (though multiple computers can be connected with network devices), and uses a co-routine model of processing. Our model integrates the network as a central object, and we use a mostly user-level implementation for maximum flexibility, portability and ease of implementation.

3.10   Plan 9 and $8\frac{1}{2}$

Plan 9 builds on UNIX Streams, but with better integration for the network and its 8 1/2 windowing system [14, 13]. Plan 9 was designed to be a distributed system, with users' local machines being just powerful enough to run the window system and do

simple interactive tasks, and most of the real work being done on more powerful remote servers. It treats all I/O as files, and uses pipes to transmit I/O commands between clients and the windowing system on the server. It can then multiplex and compose these pipes in various ways.

$8\frac{1}{2}$ is a windowing system designed for the Plan 9 operating system [13]. $8\frac{1}{2}$ provides functionality equivalent to that of X. Since all resources in Plan 9 are represented as files, $8\frac{1}{2}$ essentially works as a file server, providing client processes with files in the /dev directory that represent the mouse, screen, keyboard, and display. Each process has its own version of the files in its own namespace. Processes access the window system by reading and writing these files. A process cannot tell if it is running remotely or on a local machine; it sees the same files, regardless. Because $8\frac{1}{2}$ simulates its own environment for its clients, it can run recursively inside itself, providing an easy way to open a session on a remote machine where each process in the session is automatically opened on the remote machine. Because $8\frac{1}{2}$ is a window system, it has many of the same drawbacks X does, especially with regards to the amount of work done by the client machine. It offers advantages over X in that the mechanisms for running remote programs are even more transparent, and $8\frac{1}{2}$'s recursive abilities make it easy to open a distinct session on a remote machine. We share many ideas with Plan 9, though a key difference and central notion in our work is that of paired modules to support networked I/O with transformations.

3.11   Container Shipping

In Container Shipping, Pasquale et al describe an operating system mechanism for transferring large data between multiple processes [11]. It introduces the idea of *containers*, a set amount of shared space between processes, and *pallets*, units of data within that shared space. This separation of units of transfer and units of access is similar

to how we define the difference between containers and messages (described in Section 4.4).

3.12   Orthogonal Work

There has also been related work in the area of routing. The Click Modular Router uses the concept of modules acting on a datastream to make it easy to create routers with different functionality [7]. This work differs from ours in that it deals specifically with routing.

Remote DMA is a network protocol designed to speed up transmitting large amounts of data over the network [15]. It copies data directly into a buffer within a process's address space, avoiding multiple copies through the kernel. This work is orthogonal to our own, in that it could be combined with our work by creating Network Modules that use the protocol.

3.13   Conclusion

We share ideas with all of these systems. Like classic thin clients, our goal is to support remote I/O devices, although we wish to do it in a way that is easy to expand to new devices. Similarly to USB over IP, we want to support transparency by encapsulating the network within the device driver. Our use of modules to transform the data is reflected by many of these systems, but most classically by Streams. We combine all of these ideas, as well as the novel idea of pairing modules across the network, to develop a system that supports network transparency by hiding networking inside the driver, and uses modules to make the system easy to customize and extend.

Chapter 3, in part, has been submitted for publication of the material as "Performance Aspects of Data Transfer in a New Networked I/O Architecture" by Cynthia

Taylor and Joseph Pasquale. The dissertation author was the primary investigator and author of this paper.

Chapter 3, in part, is a reprint of the material as it appears in the article "A Remote I/O Solution for the Cloud", by Cynthia Taylor and Joseph Pasquale, which appears in the Proceedings of the 5th International Conference on Cloud Computing, Honolulu, HI, June 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 4

System Architecture

4.1   Design Goals

Our aim is for our system to meet the following goals:

Transparency: We seek to make the network as transparent as possible to both device and application, and to require no modification to the application. By moving the network functionality into the network device driver, we allow both the device and the application to act as they normally would, without being aware of the network.

Customization: We wish to support customization on many levels, whether it comes from the user, application designer, or device designer. Building our design around network device drivers provides an easy way to support this customization, as it allows us to create multiple discrete pieces that can be customized in various ways.

Modularity: Since there are a myriad of possible I/O devices, it is unrealistic to expect a single application to be able to know about all of them and be capable of forwarding their input in an intelligent manner. Device drivers are by nature modular, and we design our system to take advantage of this and to be as extensible as possible.

Granularity: A basic abstraction in our architecture is the idea of the device-application pair. We allow customization of behavior at this fine-grained level: users may wish for a device's updates to behave differently for different applications, and we
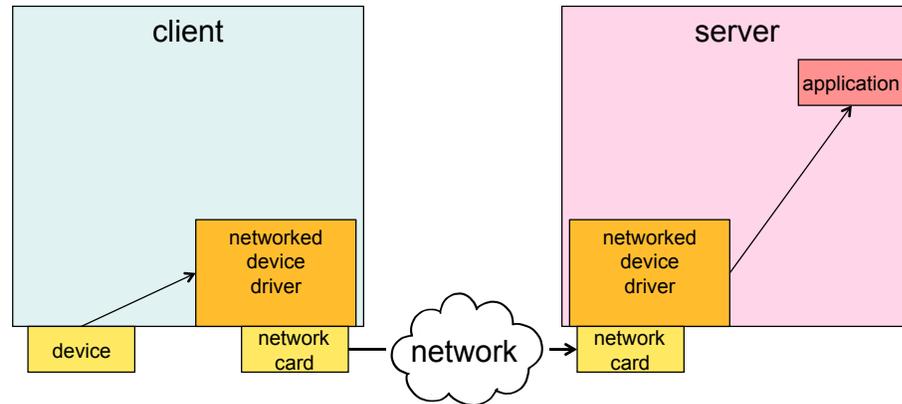
Figure 4.1. All networking is encapsulated in the networked device driver, invisible to the application and device.

support this.

Extensibility: It should be easy to add devices to the system. By making the system modular, a designer only needs to write the pieces of the system that are unique to their device, and can leverage all of the existing general pieces easily.

4.2   Architecture Summary

At its most basic level, the system architecture must support the passing of updates between a device and an application, each on a different machine, communicating over the network. Updates are created at either the device or application, passed through one half of the networked device driver, transformed in some manner and sent over the network. On the other machine, they are read from the network, the transformation may be reversed, and they are passed on to their destination (either application or device).

The system consists of four types of modules. The *application communication module* is responsible for passing data between the application and the networked device driver. Similarly, the *device communication module* passing data between the device and the networked device driver. *Network modules* send and receive updates over the network. Lastly, optional *transformation modules* modify updates as they are sent through the

system, making changes to them to compensate for the effects of the network.

When designing the architecture, we focused on our design goals. To preserve transparency, we encapsulated all network and network-related operations within a *networked device driver*, as show in Fig. 4.1. We designed our system as a series of modules to make it easy to add new pieces to the system, whether a module to support a new device, a module to add new functionality, or a module to support a new network protocol. This also allows us to support extensibility, as a designer may add whatever pieces they need easily to the system as new modules, while continuing to use all existing modules.

To support customization of functionality and make it easy to add new features and methods of processing data, we created *transformation modules*, described in Section 4.8, each of which perform exactly one operation on the data stream. To support transparency, we paired these transformation modules across the network, so when a transformation module makes a change to a message on the client, its paired transformation module on the server undoes the change, and passes the message to the application in its original form. The ability to swap out transformation modules also allows us to support granularity, making it easy to change functionality for different device-application pairings.

4.3   Data Streams

We refer to the main flow of communication in a networked device driver as a *data stream*. The data stream consists of messages that travel between the device and application, as illustrated in Figure 4.2. They travel in only one direction, being sourced at one end, and sinked at the other end. Updates are created by the device or application, and retrieved by the communication module. They are then passed through any transformation modules on the source machine, with each transformation module modifying the update in some way. The network module on the source machine then
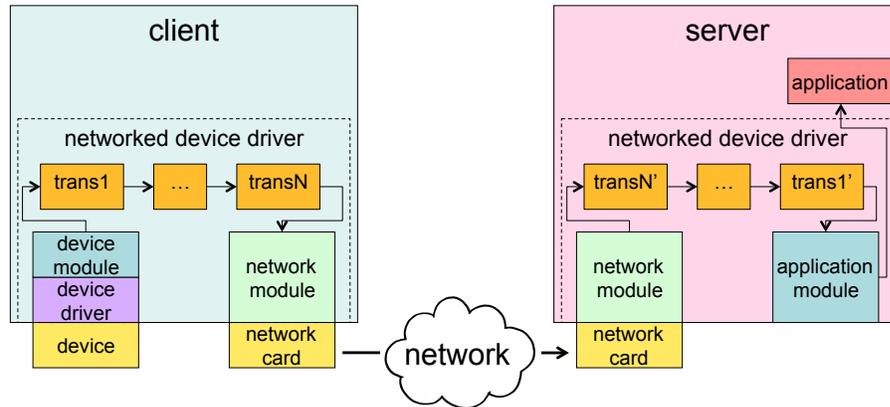
Figure 4.2. This illustrates a data stream which is sourced at the device, and sinked at the application. Messages travel from the device, across all modules, before reaching the application.

sends the updates over the network to the network module on the sink machine. They are then passed to any corresponding transformation modules, where each module has the opportunity to possibly reverse the modification applied by its matching transformation module on the source machine. The updates are then sent to the communication module, which feeds them to the sink. The system allows messages to be modified/repackaged in order for them to be effectively sent over the network, but still returned to their original form to be passed to the sink by its communication module in a network-transparent fashion.

Each networked device driver has at least one data stream. In most cases, the networked device driver will have two data streams, once sourced at the application and sinked at the device, and one sourced at the device and sinked at the application. Having application-to-device and device-to-application data streams handled separately and in a possibly asymmetric fashion allows the system to handle each data stream in a way that best fits its unique data profile. We next discuss how the various modules operate on a data stream in more detail.
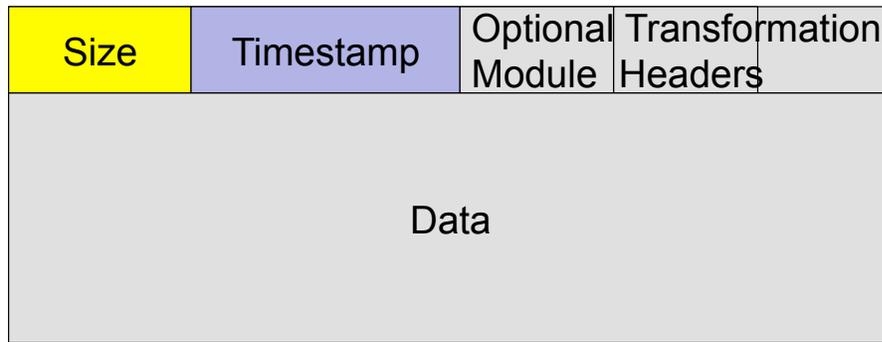
Figure 4.3. The only required header fields are size, and creation time. Individual transformation modules can add their own optional headers, which intermediate transformation modules treat as part of the data. Optional headers are stripped off by the paired transformation module on the server side.

4.4   Header Format

Each message in the I/O stream is encapsulated in a *container*, which has a short header. Fields in this header are the size of the message, and a timestamp added by the communication module when the container is created. In addition to this, individual transformation module pairs may add their own header fields after these two, as shown in Figure 4.3. Headers for specific transformation modules will be added by one half of the pair, and stripped off by the other half of the pair. Thus, they are encapsulated with the message for all transformation modules in between the pair. This allows for all of the transformation modules to access the most frequently needed information (size of message, and time created), while individual modules can add information about the message that is specific to their modification, and have it be ignored by the rest of the modules.

4.5   Device Communication Module

The function of the *device communication module* is to receive information from the raw driver, i.e. the original, unmodified device driver supplied by the device

manufacturers. To get the information from the raw driver, the device communication module interacts with the driver through its API, just as any other application would. Since it uses an API specific to the device, the device communication module must be custom-written for each device. It may wait for events raised by the device or poll, depending on how a particular API works. The device communication module operates at the user level. After the device communication module has received information from the raw driver, it forwards it to the first of the transformation modules or to the network module. Since we discuss transformation modules in a separate section, we next describe the network modules.

4.6   Network Modules

The *network modules* are a pair of modules, one on each side of the network. The role of the network module is to send data over the network: any higher level functionality, such as buffering or ordering of updates, is left to the transformation modules. The only additional work the network module does is to send an entire update to the transformation modules, even if it takes multiple reads from the network, rather than sending partial updates as it receives them.

The network modules are generic, and can be used with any device. For example, TCP may be appropriate for non real-time applications reading from video devices, where large updates may be split up into multiple packets, and it is important to receive packets in order. For devices that send smaller updates, where updates are already timestamped or ordering does not matter, UDP may be a natural fit. Modules for new or experimental network protocols may also be created.

4.7   Application Communication Module

The last transformation module passes the update to the *application communi-cation module*, which communicates with the application using the raw driver interface. Since the application communication module is based on the raw driver for the device, it must be custom-written for the device. If the raw driver is purely a kernel driver, the application communication module will consist of both a user-level component and a kernel-level component. The kernel-level component is necessary for the application to be read from the device without modification, and so we include it to support transparency. For devices with user-level drivers, the user-level component of the driver may simply be rewritten to accept input from a pipe, rather than from the raw driver.

4.8   Transformation Modules

Messages generated at either the application or device pass through a series of optional transformation modules before they reach their destination. These modules are designed to give the system the ability to add extra functionality, without losing the advantages of transparency. Each module is designed to perform a specific task, whether it is averaging messages from the device, encrypting or decrypting, or doing more complex video processing such as face finding. Modules may not require specific knowledge of the message content or format and thus are able to process any update, or, if they depend on knowing particular attributes of the message format, must be written for a single device.

The type of additional functionality required will vary for each application. Some applications only require that each update be buffered. Applications that have real time constraints may make do with only the latest updates or an average of past updates. Applications that process a lot of data may benefit from compressing information before

it is sent over the network as a performance optimization. Sensitive applications may benefit from having their information encrypted before sending. Because of this, the ideal source of information on how the networked device driver should act can be the application developers or end users.

Different functionality will require different levels of customization for the device. A function such as averaging will require knowledge of the exact message format, especially for a message which may contain multiple parts, e.g. an X coordinate, a Y coordinate, and a timestamp. However, functions similar to compression will be completely agnostic about the format of the data they are acting on, and can be used for all devices. In the middle lie functions like downsampling video, which will need to be aware of the video width, height, and general format, but can be used across multiple devices.

4.8.1 Transformation Module Pairs
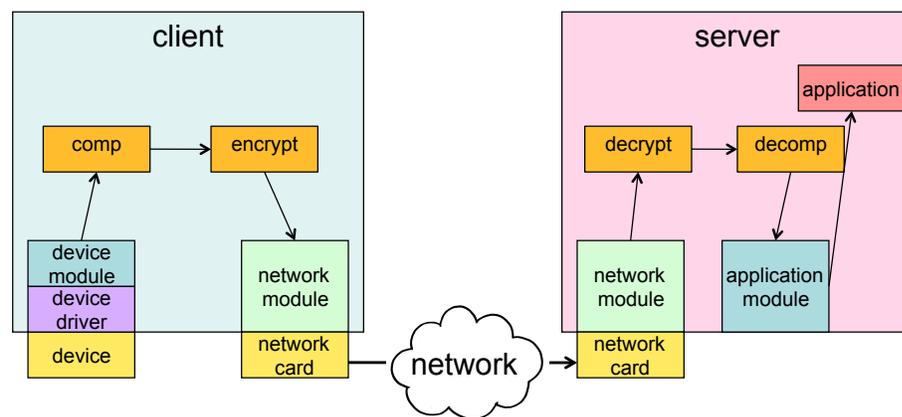
Figure 4.4. Transformation modules work in pairs, one applying an operation, such as compression, and one undoing it (i.e. decompressing). They are automatically ordered so that actions are undone in the reverse of the order in which they are performed.

To preserve transparency, any change made to the format of the message must later be undone. If a message is encrypted on the client side, it must be decrypted on the

server side before it reaches the application. Operations must be undone in reverse of how they were performed: if encryption is done before compression, then decompression must be done before decryption. Some transformation modules (e.g. averaging) change the content of the message, rather than its form, and thus may have no natural reverse of their action. Each such module is paired with a "no-op" module.

4.8.2 Functionality

We now present examples of transformation modules. These illustrate the kinds of tasks transformation modules are designed for, and what part they play in our system.

Averaging   Similar to receiving the most recent update at set intervals, some applications work best with an average of the updates received in a time period. Averaging requires knowledge of the format of the message, since many updates contain attributes that need to be treated separately (e.g. x and y coordinates, timestamps). Thus, averaging modules must be written for a specific update format.

Buffering   For devices such as video where the rate of the updates matter more than their freshness, users may want to buffer. Buffering modules work as follows: On the sink machine, the module takes as parameters a minimum and maximum number of updates to buffer. When the number of messages it is holding fall below the minimum parameter, it sends a request out-of-band to the module on the source machine, requesting the number of updates that will bring it back to the maximum. The module on the source machine releases up to that number of updates into the data stream. The module on the source machine also takes as a parameter maximum number of updates to buffer; when it reaches that limit, it discards earlier updates.

Bundling   The bundling module is designed to send several small updates as a single network packet. It takes as a parameter the number of updates to be bundled into one container. On the source side, the module collects updates until it has the required number, then packages them into one container and passes it to the next module in the data stream. On the sink side, the module unpacks the container, and release the updates inside back into the data stream using the time intervals they were originally sent at.

Compressing   Compression is useful to avoid sending a large amount of data over slow networks. Since message lengths are prepended to updates, it does not matter that compression changes the update size. We implemented compression/decompression modules using the zlib compression library.

Encrypting/Decrypting   When using sensitive applications over insecure networks, users may wish to add security by encrypting updates. Since all the module needs to know to encrypt or decrypt is the length of the message, this easily generalizes to multiple devices. The transformation module on the source side of the driver encrypts the message, and the module on the sink side decrypts it. We implemented AES encryption modules using OpenSSL.

Multiplexing   Since a key advantage of the cloud is the ability to perform distributed processing, our system needs the ability to send input from a device to multiple applications on multiple machines. We can create transformation modules which multiplex data in intelligent ways - for example, depending on how we are processing video input, we might want for all the video to go to every machine, different frames to go to different machines, or different portions of each frame to go to different machines.

Periodic Updates Some applications do not require every device update, and only need to receive the most recent update at set time periods. This transformation module takes a time constant and sends the latest update at set intervals.

Pre-Fetching For applications that request updates and exploit some sort of locality, requests for updates clustering around the current update could be artificially generated and sent to the device, with the responses stored on the server side awaiting the application's request. This requires specific knowledge of application behavior and request formats.
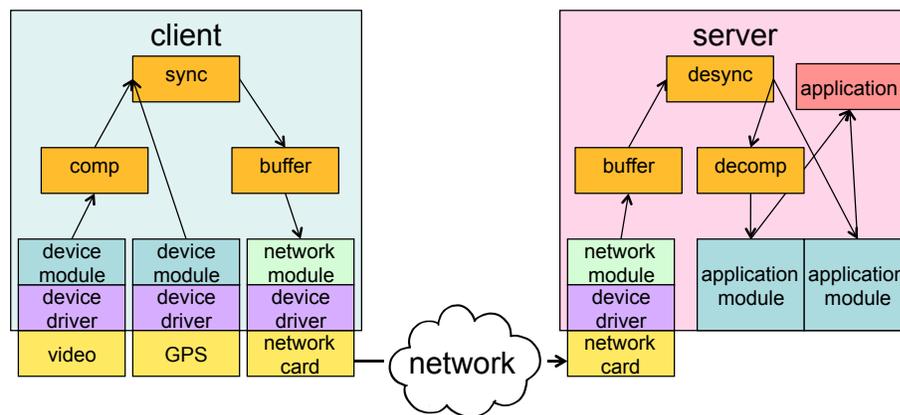
Figure 4.5. The synchronization module combines the data streams from multiple devices. Modules can be ordered so operations are applied to each device's stream separately, or the combined stream.

Synchronizing Multiple Data Streams For applications that receive data from multiple devices, it may be necessary to synchronize the updates from these devices. This may be achieved by time stamping the updates as they are generated, and ordering them by these time stamps on the sink machine. The updates may either be combined into one data stream, or continue as two separate data streams.
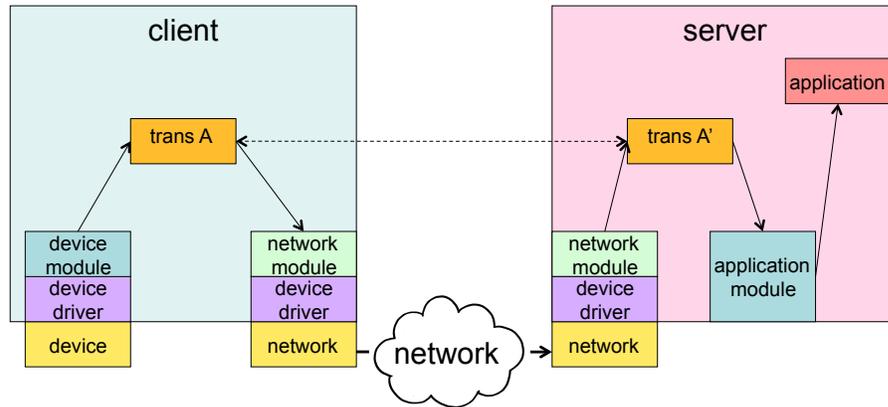
Figure 4.6. Transformation module pairs may send out-of-band messages in order to exchange control messages, which do not need to traverse the entire data stream. This avoids processing control messages, which have different characteristics and delivery needs, as though they were data messages.

Additional Functionality Applications may extend their functionality by performing modifications to the I/O stream. For example, video frames can resized, color-shifted, or even have computer vision techniques such as object-tracking added. Since these functions need to have specific knowledge about the content and format of messages, they need to be designed for a specific device.

4.9 Out-of-Band Messages

All transformation modules also support out-of-band messaging, i.e. using a separate communication channel than the one carrying the data stream. Out-of-band messages contain meta information about how the modules should process the messages being sent through their stream. For example, when a buffer transformation module on the server is running low on buffered updates, it will send a request for new updates to the buffer module on the client. This is required to enable transformation modules to respond as a pair across the network, especially when reacting to changes in network or other conditions.

When designing the system, we decided it was necessary for transformation module pairs to be able to exchange control messages directly, rather than passing them along the data stream. Messages in the data stream are passed through transformation modules, which are designed optimally process messages from the device. However, control messages between modules will have different data characteristics and needs, and should not be treated the same as data messages. For example, if the data stream is being buffered, we will not want control messages, which may need to convey information immediately, to also be buffered. Furthermore, sending control messages along the data stream would require transformation modules to check each message to see if it was a data or control message, and if it was a control message for that particular transformation module, requiring more fields added to the headers for every message. Instead, we simply send control messages directly between paired modules, allowing them to be processed for their own data characteristics, rather than that of the data messages, and avoiding the addition of complicated addressing schemes.

4.10   Conclusion

In the networked device driver architecture, we encapsulate the network and all related processing within the device driver, in order to preserve network transparency. We add transformation modules to the architecture in order to be able to process the data for the network, e.g. compressing, encrypting, buffering, synchronizing. These transformation modules are paired across the network, so that if the data is compressed on the client side, it is decompressed on the server side, preserving transparency. This pairing also implies an ordering of operations, as pairs are nested across machines: if the data stream is compressed and then encrypted on the client, on the server it will be decrypted and then decompressed. In addition to transformation modules, our system consists of device communication modules which receive data from the device,

application communication modules which pass data to the application, and network modules which send and receive data across the network.

Chapter 4, in part, has been submitted for publication of the material as "Performance Aspects of Data Transfer in a New Networked I/O Architecture" by Cynthia Taylor and Joseph Pasquale. The dissertation author was the primary investigator and author of this paper.

Chapter 4, in part, is a reprint of the material as it appears in the article "A Remote I/O Solution for the Cloud", by Cynthia Taylor and Joseph Pasquale, which appears in the Proceedings of the 5th International Conference on Cloud Computing, Honolulu, HI, June 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 5

Implementation

When implementing our system, we kept in mind our design goals, described in Section 4.1. We aim to make it easy to customize networked device drivers or create new ones. With this in mind, most of our system is implemented at the user level (i.e. outside the operating system kernel) whenever possible. We violate this only when necessary to preserve transparency. We avoid running arbitrary code in the kernel for two reasons: Buggy processes are more likely to create catastrophic system failures on errors if they are run in the kernel, and processes have the ability to maliciously affect the system if run in the kernel. Running on the user level also allows us to use all of the mechanisms provided by the kernel, such as memory copying and scheduling.

An exception is made for the application communication module, where we sometimes use kernel modules to create the device driver interface that the application is familiar with, as illustrated in Figure 5.1. Adding this kernel module at the very end allows us to preserve transparency, while still giving us all the advantages of running at the user level for the rest of the system. All device communication modules, network modules, and transformation modules run at the user level. All of our implementation is done for the Unix/Linux kernel, although our system could be generalized to any operating system.

Throughout this work, we have been extremely conscious of the tension between
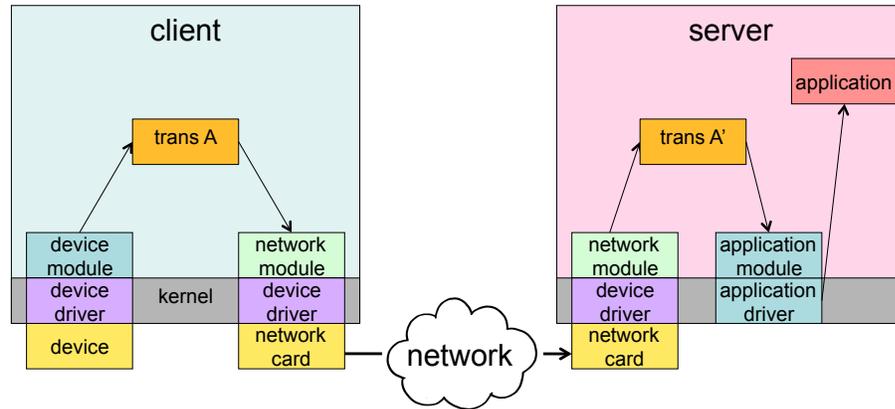
Figure 5.1. An illustration of the networked device driver architecture, with a transformation module. Only the application driver and raw driver run within the kernel, meaning only one kernel module must be created. All other modules run at user-level, allowing them to use all mechanisms provided by the operating system.

ease of creation of system modules, and the run-time performance of the system. While running modules as separate processes on the user-level gives us many advantages, it also means that we must depend on the performance of said mechanisms. We use the generic scheduler provided by the system, while a custom-built scheduler designed specifically for our system may have be able to provide better performance. While running modules as user-level processes gives us built-in modularity and makes it easy to swap out functionality, it also means that we must transfer data between separate process, which requires making calls to the kernel, adding a significant amount of overhead. These trade-offs are especially apparent in how we do data transfer, and because of that, we developed two separate system implementations, one using pipes and one using shared memory, which we compare in Section 5.1.

5.1   Data Transfer

In this section, we discuss two different implementations of our system, one of which uses shared memory for data transfer, and the other using pipes. Pipes present a

simple and easy mechanism to transfer data from one process to another. Pipes do not require any notion of an address for data: it is simply written to the pipe by one process, and read from the pipe by the next process. A messages can vary in size, and can be written/read all at once, or in sequential pieces. The process will naturally block on read calls and wait for data to become available. However, pipes require a data copy between every process, which causes a lot of overhead.

In contrast, shared memory has much less overhead, due to not requiring a memory copy between processes. However, it does not have any of the built-in mechanisms that make pipes such a natural fit for our system. In order to use shared memory without data copies, we must create a pool of shared memory between all processes in our system, and then communicate to each process what part of this memory they currently own, building a synchronization system. We must also be able to communicate what parts of the shared memory are free and can be re-used for more messages. In addition, there are issues with what happens when messages change sizes across the data stream. We describe this in detail in Sections 5.1.2 and 5.1.3.

5.1.1   Implementation with Pipes

In our pipe-based implementation, all links between modules in the data stream are created with pipes, as shown in Figure 5.2. Each transformation module has a read pipe and a write pipe. It reads a message from the read pipe, performs some transformation on the data, and then writes it to the write pipe. This is illustrated in Listing 5.1, an example of a pipes-based transformation module which reads in a message, prints it, and writes it to the next transformation module. Specifying the read and write pipe allows us to order the modules, e.g. if the compression module writes to pipe A and the encryption module reads from pipe A, messages will be compressed and then encrypted. Similarly, the device communication module, network modules, and

Listing 5.1. The body of a pipes-based transformation module which prints messages.

```
int size;
char buffer[MAXSIZE];

for (;;) {

/* read the message */
read(read_driver, &size, sizeof(int))

int received = 0;
while (received < size) {
    numread = read(read_driver, buffer+received,
            size-received);
    received = received + numread;
  }

/* print the message*/
int i;
for (i=0;i<size;i++){
    printf("%c", buffer[i]);
}

/*write the message */
  write(write_driver, &size, sizeof(int));
  write(write_driver, buffer, size);
}
```

Listing 5.2. Using pipes to connect transformation modules

```
./video_devicecommunicationmodule -w /tmp/out0
./encrypt -r /tmp/out0 -w /tmp/out1
./compress -r /tmp/out1 -w /tmp/out2 &
./networkmodule -w /tmp/out2 -i 7845
```
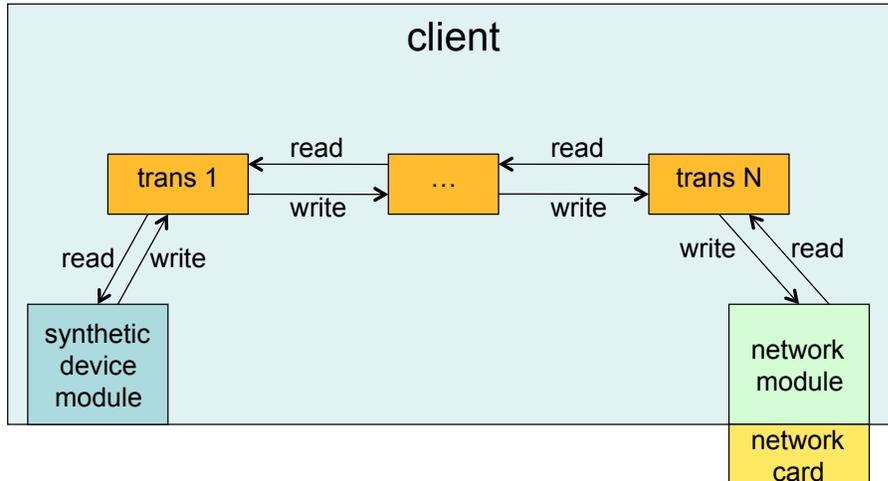
Figure 5.2. The networked device driver architecture, using pipes. Each transformation module has a read pipe and a write pipe it uses to read data from the module before it, and write data to the module after it.

application communication module communicate with the transformation modules they are connected to by reading and writing to pipes. This is shown in Listing 5.2. Each message starts with a fixed length header field containing the message content's length, so variable sized messages can be read and written within the same data stream.

5.1.2 Implementation with Shared Memory

In our shared memory implementation, a large pool of memory is simply memory mapped between all of the modules in the networked device driver. Updates are written to the shared memory by the device communication module on the device side, with multiple updates kept in the memory pool at the same time. Each update is modified by each transformation module in order, with all changes occurring within the shared memory (synchronization is done via pipes, discussed below). When the update has gone through all the transformation modules, the network module reads it from the shared memory, writes it to the network, and sends a message to the device communication module letting it know the area of memory occupied by the update is now available to be
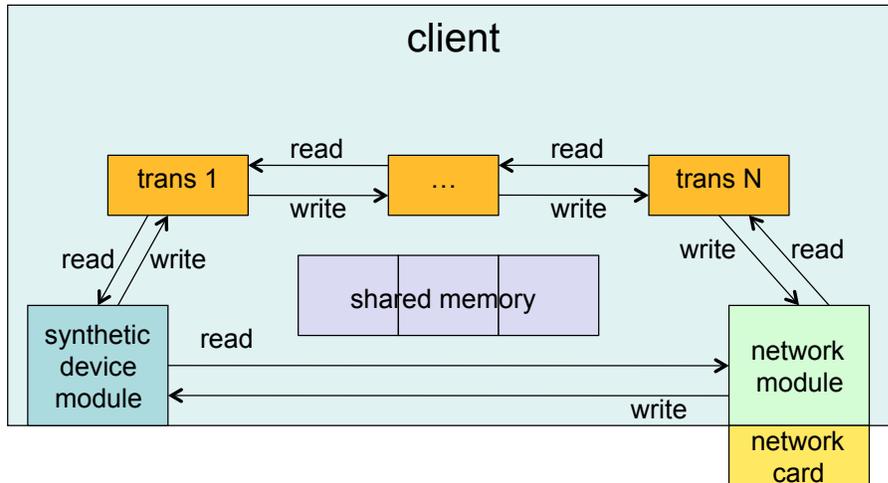
Figure 5.3. The networked device driver architecture, using shared memory. Control messages are sent through pipes, in order to let processes know which section of shared memory currently belongs to them.

written over. Similarly, on the application side, the network module reads in an update, and then writes it to the shared memory pool, where it is modified by the transformation modules. When it is read by the application communication module, the latter sends a message to the network module, letting it know that space in memory is now available.

To synchronize between the modules, we have a system of pipes similar to the implementation described in Section 5.1.1, but used purely for synchronization, shown in Figure 5.3. When a module is done processing an update, it writes the starting address and size of the update to the next module in the data stream, signaling that the next module is now the owner of that region of the shared memory. This is shown in Listing 5.3, an example of a simple shared-memory-based transformation module.

5.1.3   Using Pipes versus Shared Memory

Pipes generally provide a more natural and flexible interface than shared memory. Using pipes automatically provides support for messages that change size while traveling through the data stream, such as in compression, messages that are combined, such as in

Listing 5.3. The main loop of a shared-memory-based transformation module which prints messages

```
int start,size=0;
for (;;) {

/*read the starting location and size of the message
      in shared memory */
 read(read_driver, &start, 4)
 read(read_driver, &size, 4)

/*print message */
int i;
for (i=0;i<size;i++) {
    printf("%c", sharedspace[start + i]);
}

/* write start and size */
 write(write_driver, &start, 4);
 write(write_driver, &size, 4);

}
```

bundling, and the additional headers that are added and removed by many transformation modules. When using shared memory, the module creator must keep track of the memory used so that it can eventually be reported as free by the network module once the message has been sent to the network, and then reused by the device communication module. This becomes more complicated when the size of the message changes across modules (from when the message was created by the device communication module to when it reaches the network module).

Complexity increases when the message size increases as it traverses the data stream. This leads to two alternatives: One can allocate some maximum amount of space to each message when the device communication module creates it, and have the potential of large amounts of wasted shared memory. Or one can try to create a system where non-adjacent sections of memory belong to the same message, and deal with the complexity and overhead of keeping track of which sections are in use, and

communicating this information to all modules. Either way, it is clear that pipes are easier to use, though the question remains whether memory copying – an unavoidable result when using pipes but avoidable when using shared memory – creates intolerable overhead. We address this question in Section 6.8.

5.2   Devices

Another goal is to allow device makers, application creators, and users the extend the system as they see fit. To add support for a new device, what must be provided are a raw driver, a device communication module using the API for that driver, and an application communication module. These three components will allow the device to work with all of the pre-existing functionality provided by the transformation modules. Below, we describe three devices we built networked device drivers for, and the process of building modules for them.

5.2.1   Space Navigator

As an example of what is involved in supporting a somewhat exotic device, we created a networked device driver for a 3D mouse device called the Space Navigator [2]. The Space Navigator is a joystick designed for manipulating three dimensional objects and spaces. It produces updates with six different values: 3 translation values, and 3 rotation values. The device produces new values at a rate of 62.5 Hz.

The device communication module for the Space Navigator runs as a user-level process that receives events from the Space Navigator, processes these events into our update format, and then writes the updates to a pipe.

The native device driver that comes with the Space Navigator actually operates at user level, so to create an application communication module for it, we rewrote the device driver to receive updates from a pipe (rather than from the raw device). This was

Listing 5.4. The main loop in the keyboard device communication module

```c
while(1) {

  /* how many bytes were read */
  size_t rb;
  /* the events (up to 64 at once) */
  struct input_event ev[64];

  /*read from the evdev event file*/
  rb=read(fd,ev,sizeof(struct input_event)*64);

 /*iterate over each event we just read*/
  int yalv;
  for (yalv = 0;yalv < (int)(rb / sizeof(struct input_event));
           yalv++)
  {
      int keybd[3];
      int size[1];

      size = 3;
      keybd[0] = ev[yalv].type;
      keybd[1] = ev[yalv].code;
      keybd[2] = ev[yalv].value;

    /*write the size of the message to the write pipe*/
    write(write_driver, &size, sizeof(int));
    /*write the keyboard evdev information to the write pipe*/
    write(write_driver, &keybd, sizeof(int)*3);
  }
}
```

simple and required very few changes to the structure of the driver. Because the modified

driver continues to raise events in exactly the same way as the original, no modification

is needed for any application to use our version of the driver.

5.2.2   Mouse and Keyboard

We created networked device drivers for the basic mouse and keyboard. In Linux,

both the mouse and keyboard use evdev events, so we created modules for them that work

in very similar ways [21]. The device communication modules (which run as user level

processes) read from the drivers of the devices and capture their events, as illustrated in

Listing 5.5. The user space keyboard application communication module

```
while (1) {

  int size;
  int evdev[3];

  /*read the keyboard information from the read pipe */
  read(read_driver, &size, sizeof(int)))
  read(read_driver, &evdev, size);

   /* Send information to the kernel keyboard driver */
   sprintf(buffer, "%d␣%d␣%d␣", evdev[0], evdev[1],evdev[2]);
   write(sim_fd, buffer, strlen(buffer));
   fsync(sim_fd);
 }
```

Listing 5.6. The kernel space keyboard application communication module

```
 /* Sysfs method to input simulated coordinates to the
      virtual mouse driver */
static ssize_t write_vms(struct device *dev,
    struct device_attribute *attr,  const char *buffer,
    size_t count)
{
  int button,press,type;

  /*scan input buffer*/
  sscanf(buffer, "%d%d%d", &type, &button, &press);

 /*generate event based on input*/
  if (type == EV_KEY) {
    input_report_key(vms_input_dev, button, press);
    input_sync(vms_input_dev);
  } else if (type == EV_MSC) {
    input_event(vms_input_dev, EV_MSC, MSC_SCAN, press);
  }
    return count;
}
```

Listing 5.4. Updates with information from the events are generated and written to the output pipe.

The application communication module consists of two parts, a user level process, and a kernel module. The user level process reads in the update from its input pipe, and writes it to a sysfs node which maps to the kernel module, as shown in Listing 5.5. The kernel module then generates the evdev event for the update, shown in Listing 5.6. Because we use the kernel module to generate evdev events, the system reacts exactly as though the mouse and keyboard events had been generated by a physical mouse and keyboard.

5.2.3   Video Card

A video card is generally accessed through a frame buffer, the area of memory that holds the pixel values for the display. Its device communication module (that runs as a user level process) reads the framebuffer values, and then writes them to its output pipe. Likewise, for the application communication module, pixel values are read from its input pipe, and then written to the frame buffer of the remote display, shown in Listing 5.7.

5.3   Network Modules

We have implemented network modules for both TCP and UDP. Additional network modules could easily be written for other network protocols. The code for the network module is quite short, about 200 lines of C code. For the pipe implementation of the network driver, we simply read in from a pipe into a buffer, and then write the contents of the buffer to a socket, shown in Listing 5.8. Similarly, on the server side, we read from the network socket into a buffer, and then write that buffer to a pipe. For our shared memory implementation, we simply write to a socket from a designated section of shared memory, and similarly read from a socket into a designated section of shared

Listing 5.7. The framebuffer application communication module

```
for (;;)
{
        /* read in the size of the update , and its X and Y
        values */
        int size , X, Y;
        read(read_driver , &size , sizeof(int));

        /*X and Y are headers added by the
              device communication module */
        read(read_driver , &X, sizeof(int));
        read(read_driver , &Y, sizeof(int));

        struct iovec iov[Y];

        /* since we can display portions of the buffer , we need to
        map the beginning of each row to the correct spot in
        the frame buffer */
        for (i=0; i<Y; i++)
         {
              unsigned char *temp=fbbuf;
              temp=temp+i*vinfo.xres*cell_size;
              iov[i].iov_base = temp;
              iov[i].iov_len = X*cell_size;
         }
        /* read into the frame buffer */
        int r = readv(read_driver , iov , Y);
}
```

Listing 5.8. The network communication module for TCP

```
int size;
char buf[MAXSIZE];
for (;;) {

  read(fd_driver, &size, sizeof(int));

  received = 0;
  while (received < size) {
    numread = read(fd_driver, buf + received, size-received);
    received = received + numread;
  }

send(_s, &size, sizeof(int),0);

sent = 0;
  while (sent < size) {
    numwrite = send(_s, buf+sent, size-sent,0);
    sent = sent + numwrite;
  }
}
```

memory on the application side.

5.4 Conclusion

We created a Linux implementation of the networked device driver architecture. Each module runs as a process in our implementation, and these processes run almost entirely at the user-level. This allows us to leverage existing Linux mechanisms such as scheduling. We present two separate implementations of data-passing, one using shared memory, and one using pipes. While our shared memory implementation has less overhead, the pipes implementation is drastically easier to implement, automatically providing many of the features we must implement separately when using shared memory. We also provide examples of device and application communication modules for several devices, including the mouse, keyboard and video card.

Chapter 5, in part, has been submitted for publication of the material as "Performance Aspects of Data Transfer in a New Networked I/O Architecture by Cynthia Taylor and Joseph Pasquale. The dissertation author was the primary investigator and author of this paper.

Chapter 5, in part, is a reprint of the material as it appears in the article "A Remote I/O Solution for the Cloud", by Cynthia Taylor and Joseph Pasquale, which appears in the Proceedings of the 5th International Conference on Cloud Computing, Honolulu, HI, June 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 6

Performance

6.1   Base End-to-End Time

We first wanted a base measurement of how long it will take a message to travel along the data stream from the source to the sink, on the most basic of network device drivers. This tells us how much latency our system is adding to the device, and helps determine if our system adds an unreasonable amount of overhead.

In order to measure this, we created a basic network device driver, consisting of a device communication module, two network modules, one on each side of the network, and an application module. We measured the end-to-end time for a variety of message sizes, as our system must work for all devices, regardless of how much data they produce. In order to test this, we created a synthetic device module which produces a random character array in a specified size every two seconds. The device module paused for two seconds between sending messages to avoid any bandwidth issues from sending multiple messages at a time. When the character array was created, it was time stamped. After it had reached the application communication module, another time stamp was taken, and the first time stamp was subtracted from the second to determine the travel time of the message.

Our experiment was made more complicated because we could not synchronize
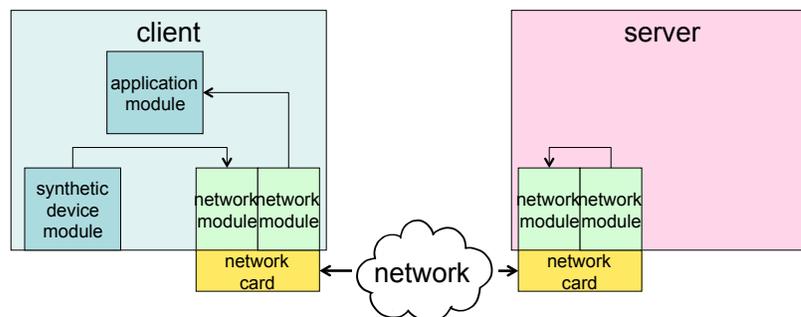
Figure 6.1. The experimental setup for our base end-to-end time. The system must be effectively doubled in order to measure results on the same clock, so there are two network modules on each machine, and the synthetic device module and application module are both located on the client.

clocks across machines with enough granularity to make our measurements. In order to get reliable times, the timestamps had to be created in both the device communication module and the application communication module on the same machine. To do this, the network device driver was essentially doubled: two network modules are created on each machine, one sending and one receiving, and the application communication module is moved onto the same machine as the device communication module. A message was created by the device communication module, sent to network module, which sent it across the network, where it was read in by one network module, forwarded the next network module, which sent it back across the network, where it was read in by the final network module, and sent to the application communication module, where the second time stamp was taken, and the round trip time calculated. This experimental setup is illustrated in Figure 6.1. To get the end-to-end times from the round trip times being measured, the times were divided by two.

All tests were performed on a two Dell Optiplex 320 machines with dual-core Intel Celeron Chips and 133 MHz FSB clocks. We used these machines because they are examples of relatively inexpensive off-the-shelf hardware. Both machines are running Ubuntu Linux. Both machines have wired connections to a relatively fast campus network,
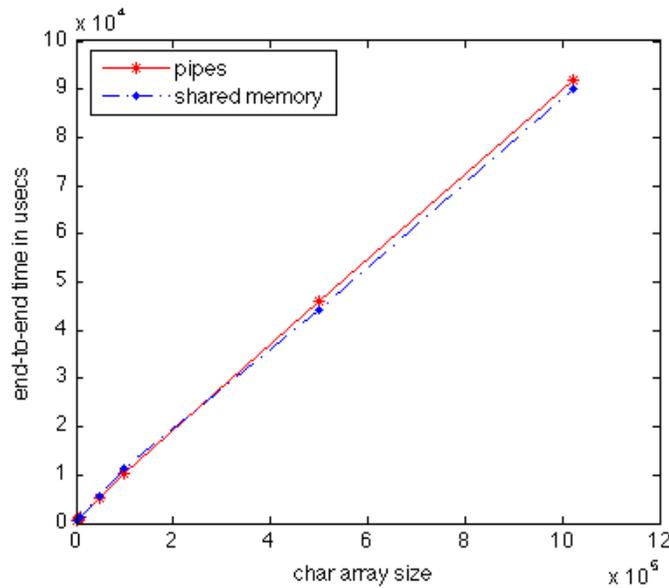
Figure 6.2. Random character arrays in different sizes are sent across the network device driver, and their traversal times are measured. Since the network device driver is doubled, the round trip times are divided by two to get the end-to-end times, displayed here.

with a sample ping round-trip time of 0.235 msec. All measurements are averaged over 450 tests, as this gaves us sufficient data points while also allowing the tests to complete in a reasonable time.

Our results are shown in Figure 6.2 and Table 6.1. For an array of 1000 characters, it took 632 usec to travel from sink to source under pipes, and 618 usec under shared memory. For an array of 1024000 characters, it took 91988 usec to traverse the network device driver wit pipes, and 90032 usec with shared memory. If we fit a line to this data, it has a slope of 0.17, so each additional character will add approximately 0.17 ucsec, a very small amount of time.

For the end-to-end times of our basic network device driver, with no transformation modules, we see that there is very little difference between the pipes implementation and the shared memory implementation. We also demonstrated that our system adds a relatively small amount of overhead, especially for smaller message sizes. For example,

Table 6.1. The average time to for a message, consisting of a random character array, to completely traverse the system in microsends. Each column represents a different array size.

Num Characters	1000	5000	10000	50000	100000	500000	1024000
Pipes	632	819	1287	5408	10320	45942	91988
Shared Memory	618	826	1270	5739	11539	44322	90032
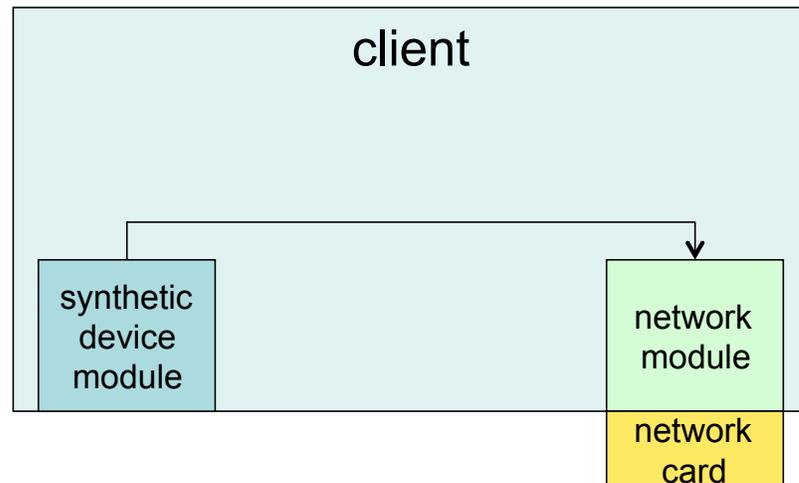
Figure 6.3. The experimental setup for end-to-end time on a single machine. Measurements are taken before the message is sent by the device communication module, and before it is sent to the network by the network communication module.

the default rate for polling a USB mouse in both Windows Vista and Ubuntu Linux is 125 Hz, or every 8000 usec. At the base rate of 632 usec for a 1000 character array (much larger than the data from a mouse), adding a network device driver would not add a significant delay to a USB mouse.

6.2   End-to-End Time Across a Single Machine

We also wanted to look at the end-to-end time across a single machine, i.e. the time it takes from an update to be generated until it is sent across the network. Looking
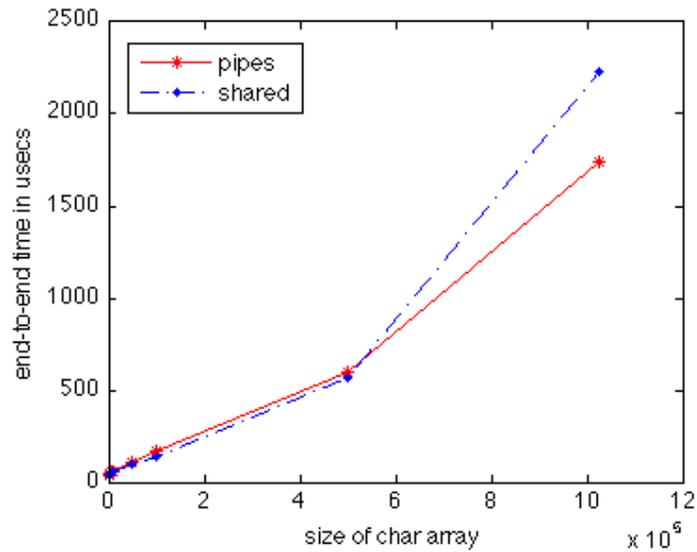
Figure 6.4. The time it takes for a message, consisting of a random character array, in various sizes, to be sent across a single machine. The time measured is from when the message is generated, to when it is about to be sent to the network. Measurements are in microseconds.

at this gave us a better idea of the performance of the system due to memory copies, as this metric will not have the slowing effects of the network.

In order to measure this, we created a basic device driver, as shown in Figure 6.3. A character array is created in the synthetic device module, and a timestamp is added to it. This character array is sent to the network module, where a second time stamp is taken right before the message is written to the network socket. The first time stamp is subtracted from the second, giving us the time the message spent within the networked device driver on the client machine, from creation to send time. There is a pause for 2 seconds between sending each array, to avoid having multiple messages in the system at once. All results are averaged over 450 messages.

Our results are shown in Figure 6.4 and Table 6.2. For a 100 character array, it took 46 usec to cross the machine using pipes, and 41 usec using shared memory. For a 1024000 character array, it took 1742 usec using pipes, and 2233 usec using shared

Table 6.2. The average time it took a message, consisting of a random character array, to traverse a single machine, in microsends. Each column represents a different size of character array.

Chars	100	500	1000	5000	10000	50000	100000	500000	1024000
Pipes	46	34	32	58	56	106	173	597	1742
Shared	41	35	43	42	56	99	137	564	2233
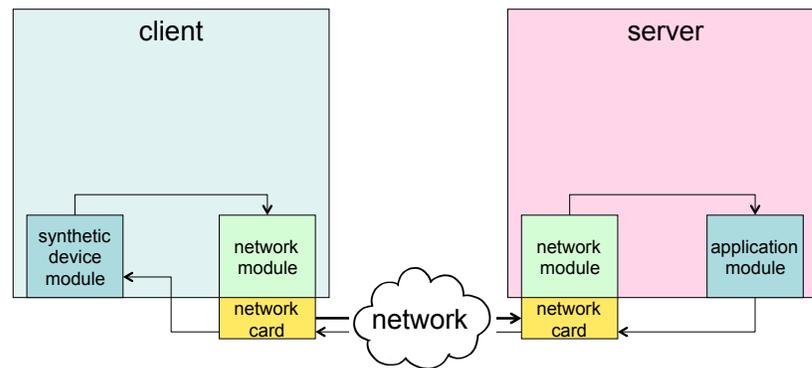
Figure 6.5. The experimental setup for our base bandwidth measurements. We add an out-of-band link between the application module and device module, in order to make measurements on the same computer clock.

memory. For most message sizes, the time added by data transfer across a single machine is less than a millisecond, showing that our system is adding very little overhead. In addition, we see that with no transformation modules, the memory copy overhead of pipes and shared memory is very similar.

6.3   Base Throughput

We next wanted to find the base throughput of our system, in order to make sure bandwidth will not be the limiting factor when dealing with devices which output large amounts of data. This is important for dealing with devices which produce large updates.

In order to measure the bandwidth of the system, a basic network device driver was created with a synthetic device communication module which produces a random

Table 6.3. The throughput of the system, in Megabytes/second. We compare the throughput of pipes and shared memory.

Pipes	10.5004 MB/sec
Shared Memory	11.0023 MB/sec

character array of a given size, as in Section 6.1. Once again, the issue of having two separate computer clocks had to be dealt with. Unlike in our previous experiment, simply doubling the device driver would not work, as this would change the amount of information traveling through the system. Instead, an out-of-band connection was added between the device communication module and the application communication module. A large amount of data was sent through the network device driver, and once the application module had received it all, it sent a small acknowledgement message back to the device communication module. The device communication module created a large character array, took a time stamp, sent the array, waited for the acknowledgement from the application module, took a second time stamp, and subtracted the first time stamp from the second to find the time it took to send the data. This is illustrated in Figure 6.5.

Testing just the network speed while running these experiments, we got a base time of 11.3 Megabytes/second when using rcp to download a large file between the two machines. We got very similar times with our system, with throughput of 10.5 MB/s with pipes, and 11.0 MB/s with shared memory. This indicates that the throughput of our basic system is largely dependent on the throughput of the underlying network.

6.4 Update Speed of Networked Device Drivers Compared to Standard Drivers

We next wanted to see how much overhead our system adds when used with an actual device, to get a more realistic picture. We wanted to make sure that our system
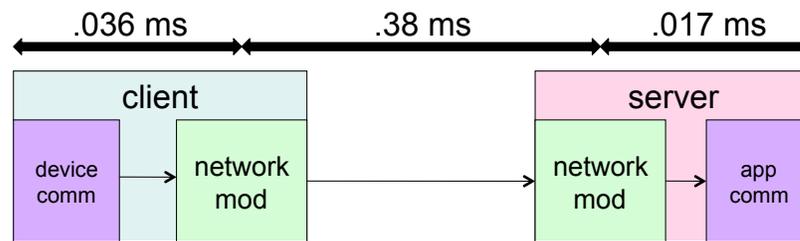
Figure 6.6. The results of instrumenting the Spacenav device driver. The majority of overhead is traveling over the network, with very little overhead due to computation.

does not add enough overhead to interfere with the device and cause significant lag. Here we use the Space Navigator, described in Section 5.2.1.

We developed a device driver for the space navigator, with a device communication module, network modules, and application communication module, and instrumented them in order to measure the time it takes an update to travel through the system. The system needed to actually forward device information from the client to the server, so the system cannot simply be doubled, as in our previous experiment, described in Section 6.1. Instead, since the update takes a one way path over the network, the system was measured in three discrete parts in order to get an accurate measurement. All measurements were taken using the rtdsc and rtdscl registers to determine the number of clock cycles that had passed, and converting from clock cycles into milliseconds. All measurements were averaged over 2000 tests, in order to achieve statistical signficance. All measurements used the pipes implementation.

Our results are shown in Figure 6.6. On the driver side, the time between when the device communication module received an update, and when the network module was ready to send it over the network was measured: this took an average of 36 usec, with a standard deviation of 9 usec. For the network, it was necessary to measure a round trip time in order to make accurate measurements using the same computer clock. We took a round trip measurement of the time it took the network module on the driver side
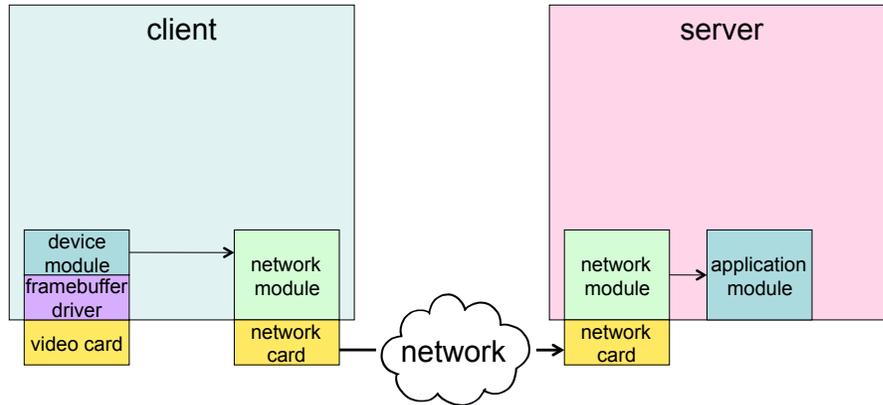
Figure 6.7. The design of the networked device driver for the video card. The application module is instrumented to measure the time between when it receives frames, which includes the time to display the frame.

to write an update to the network, the network travel time to the application side, the application communication module reading the update and immediately writing it back to the network, the travel time back to the driver side, and the device communication module reading the update. The resulting measurement was divided by two to get the one-way time, resulting in an average time of 380 usec with standard deviation of 14 usec. (A ping taken at the same time had a one-way average time of 120 usec.) On the application side, the time from right after the network module had read the update to the time when the application communication module generated an event to the application was measured, for an average time of 17 usec with standard deviation of 15 usec. The total time that our system added to the update was 433 usec.

The Space Navigator operates as a rate of 62.5 Hz, or every 16000 usec, so 433 usec would not add a perceptible delay. This shows us that when operating as a real, usable system with an actual device, the network device driver architecture does not present an obstacle to use by adding an overwhelming amount of overhead.
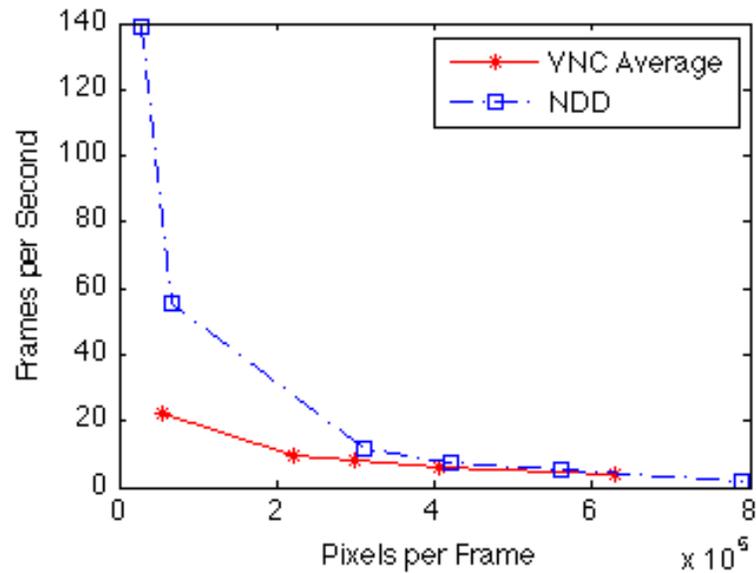
Figure 6.8. The frame rate of the video card networked device driver compared to VNC. The two send rates are very similar, with the networked device driver performing as well or better than VNC. VNC is limited to 24 frames per second at best, due to only sending frames when there is a change in the pixels.

Table 6.4. The average inter-frame time for VNC in microseconds, along with the average number of pixels forwarded per frame.

Average Pixels	55480	222918	297620	404672	627935
VNC interframe	44500	107786	127290	167625	246812

6.5   The Networked Device Driver Compared to VNC

We wanted to compare our system against custom device-forwarding applications, to show how we perform against current state-of-the-art applications also designed to solve the problem of remote I/O. We wished to show that our system performs as well as custom solutions designed for specific devices. To do this, we compared our system against VNC, the class thin client that is designed to forward video, keyboard input and mouse. We compared purely against the video forwarding capacity, comparing the frame

Table 6.5. The average inter-frame time for the video card networked device driver, with and without compression, in microseconds.

Pixels	28000	66000	312000	420000	560000	789504
Networked Device Driver	7209	18040	87434	132071	190754	542916
NDD with Compression	11598	25275	105541	184731	285813	494565

rate of a networked device driver built for the video card to the frame rate of VNC.

Using the pipe-based implementation, we built a basic networked device driver for the video card, shown in Figure 6.7. Both the network device driver and an implementation of Tight VNC[1] were instrumented to record how long it took to receive and display each frame. After initialization the VNC Client goes into a loop where it reads and processes an update, sends a request to the server, and then waits for the next update. The client was instrumented to measure how long it takes for the client to go through each iteration of this loop. Similarly, the application communication module for the Video Card runs in a loop where it reads in an update, and writes that information to the framebuffer. Code was added to this module to measure how long it takes for each cycle of reading and updating. Their performance was measured displaying a video of the television show Leverage, running at 24 frames per second. All measurements were averaged over 2000 updates.

In Fig.6.8 we compare the frames per second for different average pixels per frame being transmitted by both systems. Since VNC transmits less than the entire frame when not all pixels have changed between frames, this is a lower value than the frame size. For lower average pixel values, in the forty-five to fifty-five thousand pixel range, the networked device driver sends an average of 139 frames per second, while VNC sends an average of 23 frames per second. This is due to VNC only sending updates when there is a change in the pixel values, limiting it to the video's frame rate. However,
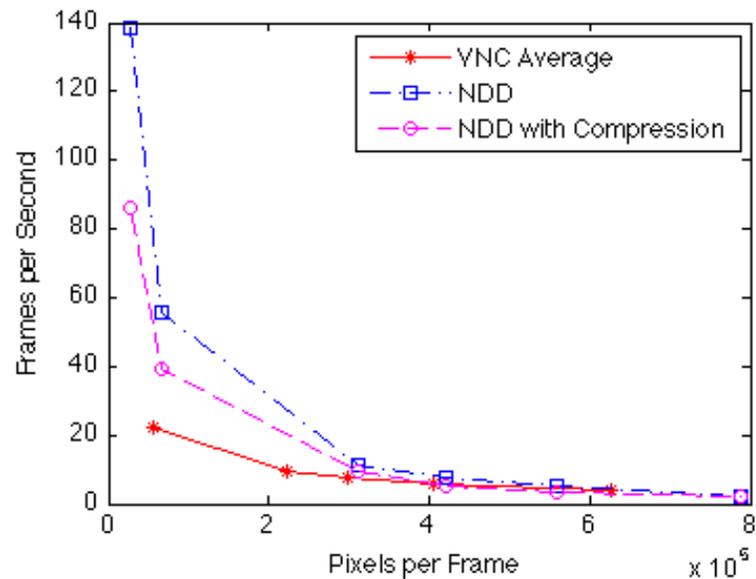
Figure 6.9. The video card networked device driver, with and without compression, compared to VNC. Even with an added compression module, the networked device driver performs as well as VNC.

when the update rate drops below frame rate at around 200,000 pixels, we see that the networked device driver sends slightly more frames per second than VNC. Our results are displayed in Tables 6.4 and 6.5.

As the average numbers of pixels per frame are increased, the networked device driver keeps pace with VNC, generating as many or more frames per second. The performance of our generic networked device driver is on par with that of VNC, an application specifically designed and optimized for forwarding video card data. Our basic implementation, even using pipes, is capable of performing on par with a commercial application designed for a specialized remote I/O task.

6.6   Adding Transformation Modules

A key component of our system is the ability to add transformation modules to process updates for the network. We wanted to show that adding a transformation module,
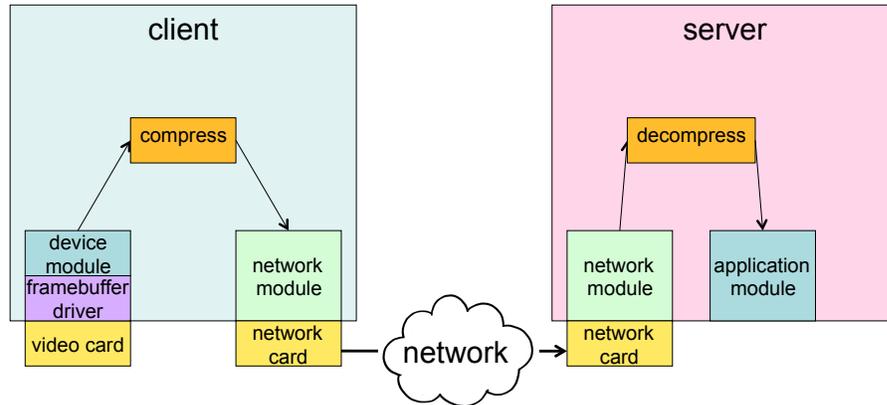
Figure 6.10. The design of the networked device driver for the video card, with compression. A compression transformation module is added between the device communication module and network module on the client side, and a decompression transformation module is added between the network module and application communication module on the server side.

a typical use cause, will not cause the system's performance to degrade drastically, or cause it to perform worse than VNC.

The networked device driver running the compression module (shown in Figure 6.10) was compared to the networked device driver without it (shown in Figure 6.7). The purpose of this experiment is not to demonstrate speed-up from compression: we are running on a fairly high-bandwidth network, and not compressing significantly enough to cause significant improvement. Instead, we wished to demonstrate that the memory copying and added computation from compression does not cause a significant detriment to performance.

At twenty-eight thousand pixels per frame, on the lower side of the spectrum, the networked device driver with compression has a lower frame rate than the one with compression, at 86 frames per second versus 139 frames per second. However, it is still significantly above the actual frame rate of the video, at 24 frames per second. Once the frame rate of both versions of the networked device driver is below the frame rate of the video, at three hundred thousand pixels, the difference in frame rates remains below
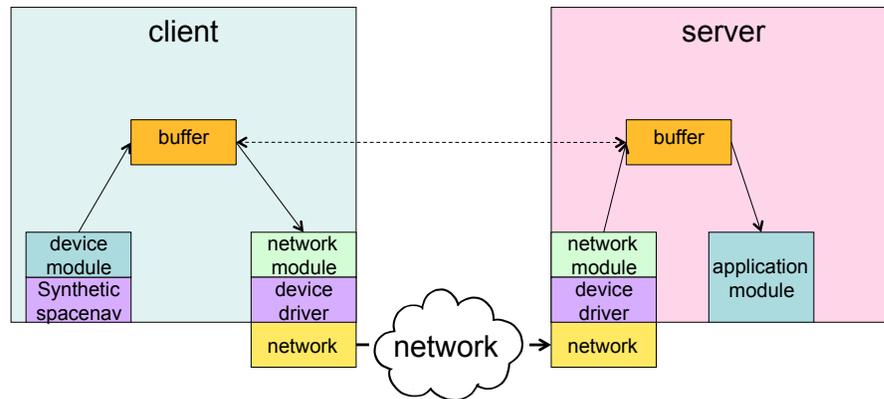
Figure 6.11. The design of the buffering experiment. Buffering modules are added on each side of the network. They exchange out-of-band control messages, so the buffering module on the server side can request new updates when it is running low.

two frames per second. The frame rate of the networked device driver with compression remains consistent with VNC's frame rate. These results demonstrate that adding a transmission module to the networked device driver does not cause a significant drop in performance, even in our pipe-based implementation.

6.7 Performance of Transformation Modules

We wanted to demonstrate that using specific transformation modules is effective in counter-acting negative performance effects due to the network. In this section, we present results from using two different modules, buffering and bundling, to counter network effects (in this case, jitter). While the use of buffering to counter-act jitter is already well-known, we present these results to show the ability of the system, configured with specific modules, to meet our goal of transparently counter-acting the temporal effects of the network on the update rate.
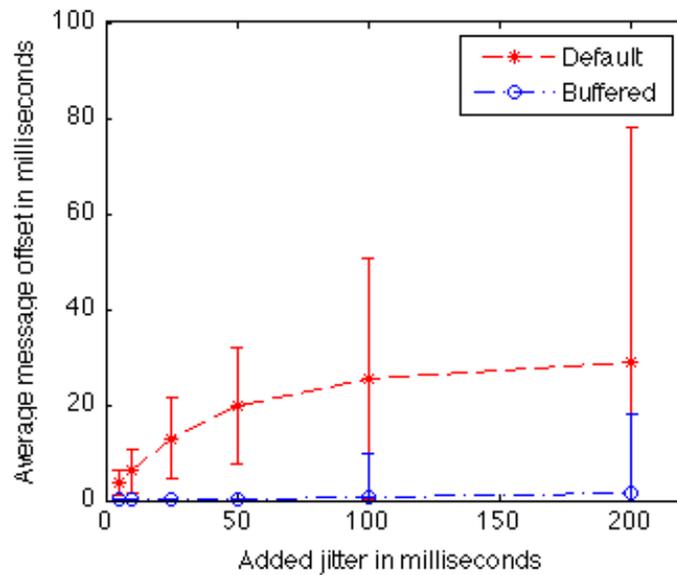
Figure 6.12. The average difference between creation inter-message time and received inter-message time, in msec. The standard deviation is show as bars. Using the buffering module drastically lowers the effects of jitter on the system. This demonstrates the buffering modules' ability to recreate the original timing between messages, due to messages being timestamped on the client side.

6.7.1 Buffering

To demonstrate our buffer modules' ability to recreate the timing of updates across the network, two network device drivers were created, one with the buffer modules, and one without. A version of the spacenav driver that automatically generates updates every 16 ms was used. This was read from by the device communication module, as normal. It was then passed to the buffering module, and then to the network module and on the sink side, it was read by the network module, passed to the buffering module, and then to the application communication module. This setup is illustrated in Figure 6.11. The sink buffer module was set to have a maximum buffer of 10 updates, and the source side buffer to have a maximum buffer of 400 updates. The inter-message distance is measured in the application communication module. It had to be measured here instead of in the application because the application communication module strips off timestamps before
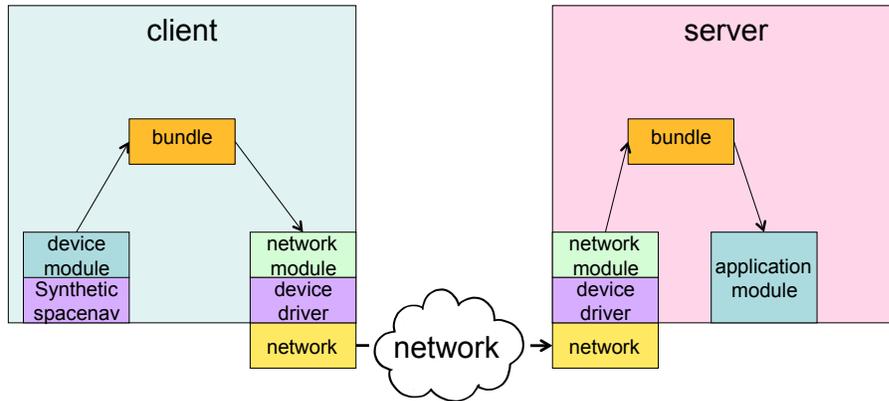
Figure 6.13. The design of the bundling experiment. The bundling module on the client side collects messages 10 and sends them together as one message, while the bundling module on the sever breaks up the message, and sends the collected messages one at a time, recreating the original timing between them.

passing an update to the application, in order to preserve transparency. The results from our buffered network device driver are compared against a network device driver which is set up exactly the same, except without the bundling module. Both network device drivers use TCP-based network modules.

To measure the jitter in both systems, the inter-message time for the updates when they were generated were compared to the inter-message time when they were received. Since each update is timestamped by the device communication module when it is generated, we were able to create an application communication module for testing purposes which generates a timestamp when it receives a new update, uses that to compute the time between it and the last update received, and compares it to the time between their generation timestamps. Our jitter metric was the difference between the inter-message times at generation and on receiving.

We ran this experiment over different amounts of network jitter varying from 5 ms to 200 ms, created using netem. Netem created this jitter by delaying sending each packet for a random number of milliseconds between 0 and N, where N is the specified maximum. All measurements are averaged over 200 runs. Our results are shown in

Figure 6.12. As shown in the figure, under the buffered system, the inter-message time at receiving remained very close to the timing when the updates were originally sent. At 200 msec of jitter, the difference was still averaging less than 1.5 msec, and until it the jitter passes 50 msec, it was averaging less than 0.1 msec difference. In contrast, without the buffering the difference between timing at send and receive immediately becomes apparent, with just 5 msec of jitter causing an average 3.6 msec time difference, and going all the way up to an average of 28.8 msec of difference at 200 msec of jitter. The standard deviations also remain much lower under the buffered system: it is under 1 msec with 50 msec of jitter with buffering, while without buffering it is already at 12 msec. This demonstrates that the use of the buffering module within the networked device driver architecture is effective in counter-acting network jitter.

6.7.2   Bundling

To show that our bundling module also prevents jitter, an experimental setup was created that is the same as in Section 6.7.1, except using a bundling module instead of a buffering module, as illustrated in Figure 6.13. Again, two identical network device drivers were set up, one with a bundling module and one without. The application communication module was instrumented to measure the difference between inter-message time on sending, and inter-message time on receiving. It was tested by using netem to add 5, 10, 25, 50, 100 and 200 ms of jitter, averaging each test over 200 trials.

As shown in Figure 6.14, with the bundling module our inter-message time stayed constant between sending and receiving, while without it quickly started to vary greatly. With bundling, the average difference in inter-message time stayed under 1 msec until over 100 msec, while without it the difference was already up to an average of 25.6 msec at 100 msec of jitter. The standard deviation stayed low with the bundling module also, showing that the inter-message time wasn't varying greatly. Standard deviation stayed
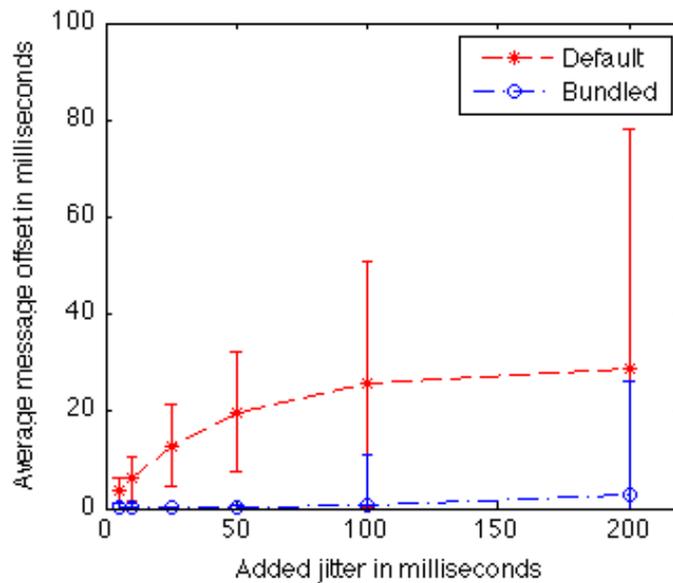
Figure 6.14. The average difference between creation inter-message time and received inter-message time, in msec. The standard deviation is show as bars. The bundling module is able to recreate the original send times between messages, despite added network jitter.

below 1 msec on with the bundling module until over 50 msec of jitter, while without the

bundling module it was already at 12.3 msec at that point.

This demonstrates that using our bundling module is effective in counter-acting

jitter in the network. With both the buffering and the bundling module, we demonstrate

the utility of using specific transformation modules with the networked device driver

architecture, and the utility of processing the data stream with transformation modules.

6.8   Effects of Adding Transformation Modules with both Shared Memory and Pipes

Since our design adds functional modules as processes, we needed to make sure

that adding these modules does not significantly impact performance. This is especially

true of our pipes implementation, where each added module will require another memory
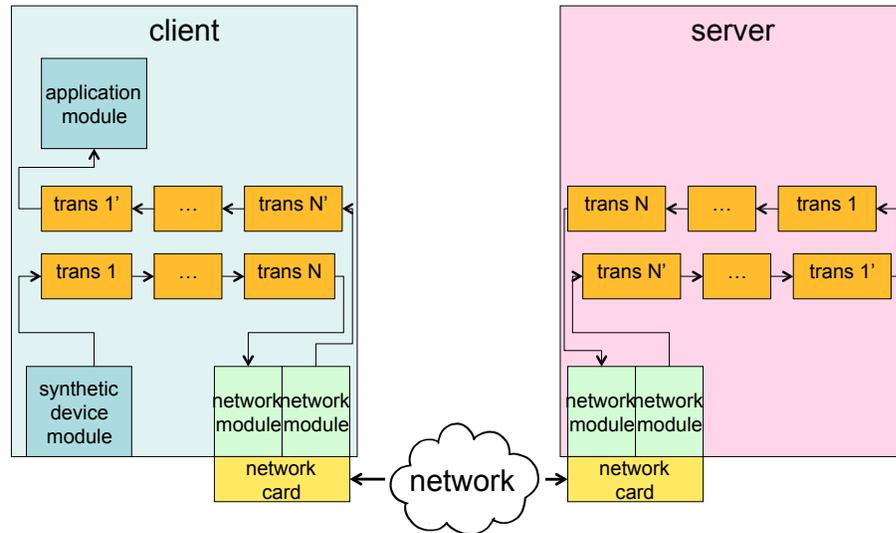
Figure 6.15. The experimental set up. The network device driver, including transformation modules, is effectively doubled, so start and end measurements can both be taken on the same machine. Measurements start in the synthetic device module, and end in the application module.

copy. In this section, we performed a number of experiments across different numbers of transformation modules with both shared memory and pipes, in order to determine the performance cost of adding a transformation module under both implementations.

6.8.1 Transformation Modules and System End-to-End Time

We first wished to determine what effect adding a transformation module pair has on the time it takes for a message to cross the entire system. This told us exactly how much extra time we add to each message with new transformation. We measured this for varying numbers of transformation modules, for both shared memory and pipes.

As in the experiments described in Section 6.1, all measurements had to be taken on the same machine. To get around this issue, an experimental setup was created, illustrated in Figure 6.18, in which the network device driver is essentially doubled. Each machine had two network modules, one sending and one receiving, and twice the number of transformation modules. This allowed both the device communication module,
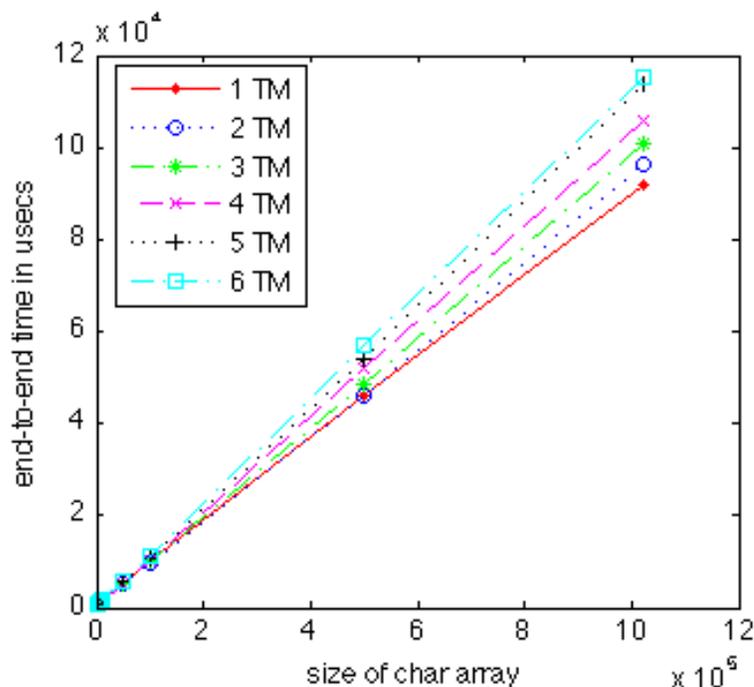
Figure 6.16. End-to-end times for pipes, across the entire system, in microseconds. Each transformation module adds an average of 6.6 percent of overhead.

where the message originates, and the application communication module, where we take the final timestamp, to be on the same machine, and thus the same clock. To get the end-to-end time for a single networked device driver, the round trip time was divided by two. This may give higher measurements than in the actual system, as more processes are running on each machine than would be running in a single version of the networked device driver. Both machines were running in single core mode. All results are averaged over 450 runs. The round trip ping time between the two machines was an average of 226 usec, giving a one-way time of 113 usec.

As shown in Figures 6.16 and 6.17, and Tables 6.6 and 6.7, our end-to-end times for the entire system clustered relatively closely together, especially compared to the end-to-end times for a single machine (shown in Table 6.8, Table 6.9 and Figures 6.19 and 6.20). This tighter clustering is most likely due to the performance the network,
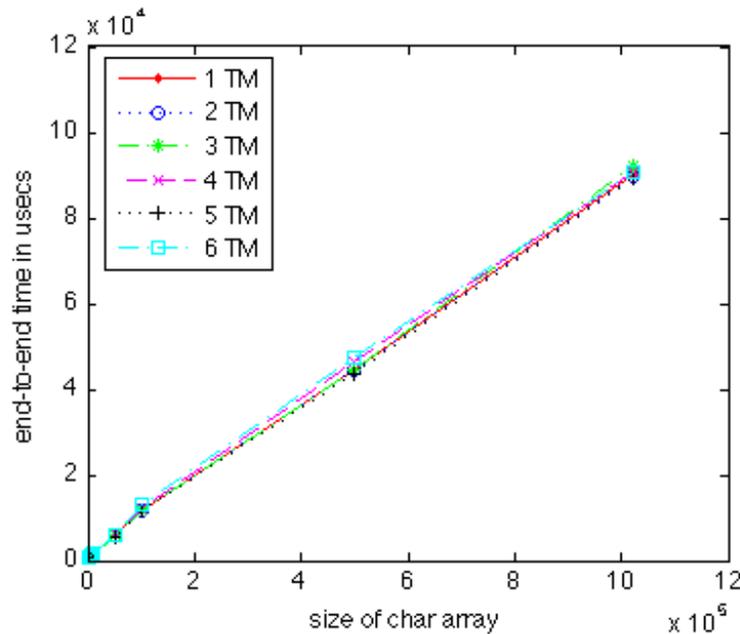
Figure 6.17. End-to-end times for shared memory, across the entire system, in microseconds. There is much less impact from adding a transformation module when using shared memory than there is using pipes (see Figure 6.16).

which will be the same in both networked device drivers. For a 1000 character array, it took 632 usec to completely traverse the system with no transformation modules, and 837 usec to traverse it with 6 transformation modules, under the pipes implementation. Under the shared memory implementation, it took a 1000 character array 618 usec with no transformation modules, and 818 usec with 6 transformation modules. It took a 1024000 character array 91988 usec to complete the pipe system with no transformation modules, and 122761 usec with 6 transformation modules. With the shared memory implementation, a 1024000 character array took 90032 usec with no transformation modules, and 91168 usec with 6 transformation modules.

Under our shared memory implementation, adding a transformation module added 1.9 percent to the latency of the entire system. Using pipes, each transformation module added an average of 6.6 percent of overhead. The pipes implementation was 1.09 percent faster than the shared memory implementation with no transformation modules,

Table 6.6. The average time to completely traverse the system in microsends under pipes. Columns are grouped by the size of the character array. Rows are grouped by the number of transformation modules.

Array Size	1000	5000	10000	50000	100000	500000	1024000
0 Trans Mods	632	819	1287	5408	10320	45942	91988
1 Trans Mods	636	862	1326	5348	9741	46190	96547
2 Trans Mods	675	904	1374	5320	9980	48432	100604
3 Trans Mods	699	940	1432	5438	10130	52189	105845
4 Trans Mods	746	985	1483	5666	10916	53984	113937
5 Trans Mods	774	1028	1532	5850	11452	56794	115333
6 Trans Mods	837	1322	1851	6070	11890	64064	122761

but each transformation module added an average of 3.41 percent of overhead to the pipes implementation compared to the shared memory implementation. We also saw that the speed of the system is heavily dependent on message size. A networked device driver with 1000 character messages and 6 transformation modules had an end-to-end time of only 837 usecs, while a device with 1024000 character messages had an end-to-end time of 91988 usecs with no transformation modules.

Our experiments show that while using shared memory offers significant performance advantages for networked device drivers with large message sizes and a larger number of transformation modules, the pipes implementation performs similarly to shared memory for networked device drivers with a smaller number of transformation modules, and smaller message sizes.

6.8.2   Transformation Modules and Machine End-to-End Time

We next wanted to look at the end-to-end time on a single machine, without the effects of the network. This let us know more about the direct impact of pipes versus shared memory on performance, without the smoothing effect of the network.

Table 6.7. The average time to completely traverse the system in microsends under shared memory. Columns are grouped by the size of the character array. Rows are grouped by the number of transformation modules.

Array Size	1000	5000	10000	50000	100000	500000	1024000
0 Trans Mods	618	826	1270	5739	11539	44322	90032
1 Trans Mods	661	856	1310	5766	11640	44586	89827
2 Trans Mods	690	889	1344	6064	11694	44733	92354
3 Trans Mods	723	946	1381	6596	11924	46218	90688
4 Trans Mods	754	1190	1410	5515	11741	43680	89388
5 Trans Mods	790	1216	1454	5915	13217	47402	90729
6 Trans Mods	818	1024	1474	6012	11750	45184	91168

Table 6.8. The average single machine end-to-end time in microsends under pipes. Columns are grouped by the size of the character array. Rows are grouped by the number of transformation modules.

Size	100	500	1000	5000	10000	50000	100000	500000	1024000
0 TM	46	34	32	58	56	106	173	597	1742
1 TM	56	59	58	69	86	176	344	1259	3797
2 TM	85	87	82	98	107	242	459	2117	5571
3 TM	84	100	88	117	142	308	605	2830	7872
4 TM	110	108	120	127	158	400	720	4114	9680
5 TM	123	127	119	150	188	460	918	5264	11593
6 TM	150	137	148	168	219	514	1019	6495	14034
7 TM	156	158	163	195	244	605	1190	7303	15555
8 TM	181	166	181	202	246	672	1375	8342	18376
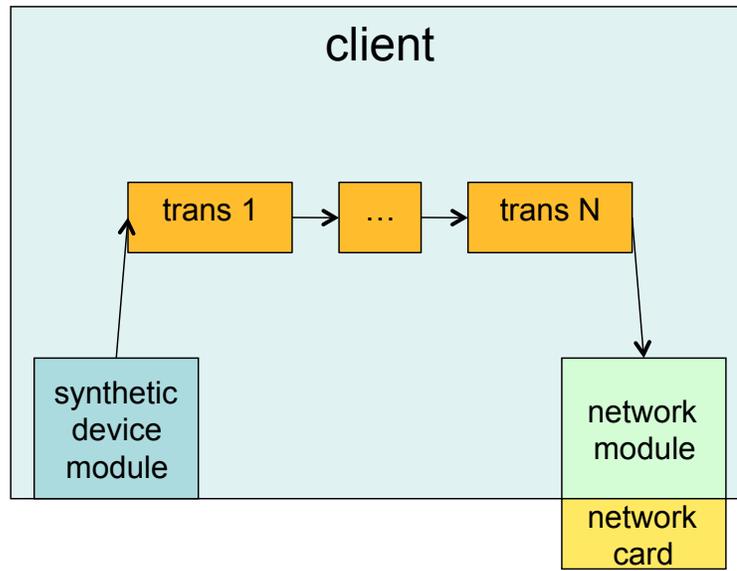
Figure 6.18. The experimental set up. Measurements start in the synthetic device module, and end in the network module. This allows us to measure how overhead differs between pipes and shared memory, without the effects of the network.

As in Section 6.2, we measured the time from when a message is generated, to when it is sent over the network, within a single machine. In order to calculate this, a timestamp was taken immediately before the message was written to the write pipe for the device communication module, another timestamp was taken immediately before the message was written to the network by the network communication timestamp, and the first timestamp was subtracted from the second. A synthetic message was generated, consisting of a large character array, and the device communication module paused two seconds between sending each message, in order to make sure multiple messages are not in the system at the same time. This experimental design is illustrated in Figure 6.18. All measurements were taken in single core mode. Each measurement was averaged over 450 trials.

Our results are displayed in Tables 6.8 and 6.9 and Figures 6.19 and 6.20. For a character array of size 100, it took 46 usec to travel through the pipe-based system,
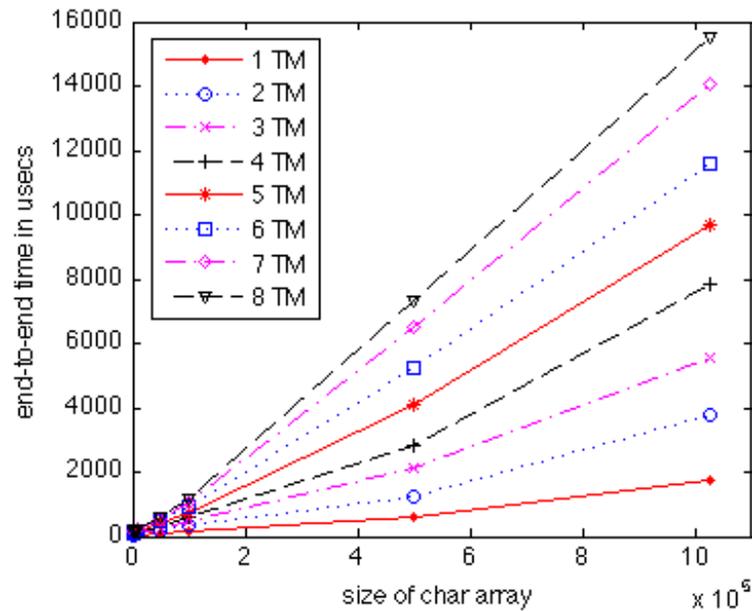
Figure 6.19. Time from source to sink on a single machine, in microseconds, using pipes. The overhead from adding a transformation module under pipes is much clearer than when looking at the whole system.

and 41 usec to travel through the shared-memory-based system, with no transformation modules. With 8 transformation modules, it took a 100 character array 181 usec to travel through the pipe-based system, and 195 usec to travel through the shared-memory based system. When transferring such a small amount of data, the performance of the shared memory and pipes is very similar. Since our shared memory implementation uses pipes for its control path, its performance will be very similar to the pipes implementation when transferring very small messages.

For a 1024000 character array, it took 1742 usec to go through the pipe-based system, and 2233 usec to go through the shared-memory based system, with no transformation module. With 8 transformation modules, it took a 1024000 character array 18376 usec to go through the pipe-based system, and 2279 usec to go through the shared-memory based system. At this larger message size, the performance differences between pipes and shared memory are much more apparent.
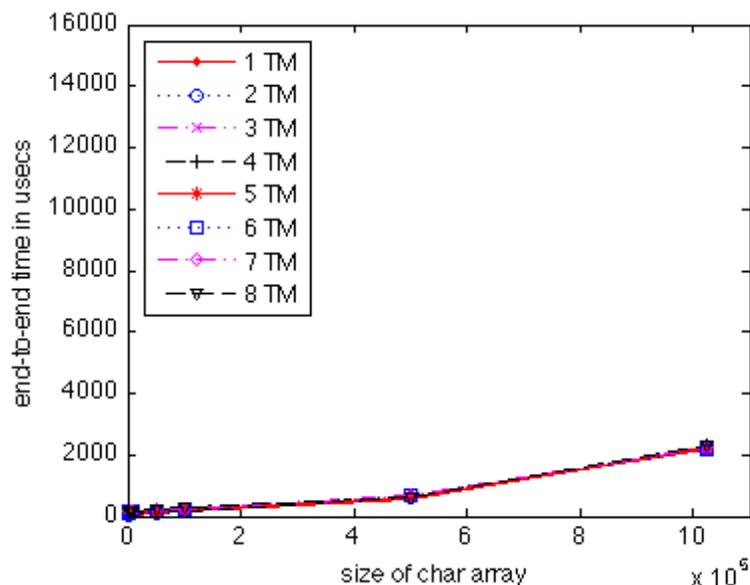
Figure 6.20. Time from source to sink on a single machine, in microseconds, using shared memory. Adding a transformation module using shared memory adds very little overhead, as opposed to using pipes.

In our single machine measurements, the performance difference between pipes and shared memory was much more apparent. Each transformation module added an average of 33.5 percent of overhead to the pipes implementation over the overhead of the shared memory implementation, as opposed to 3.41 percent in our whole-system measurements. The addition of the network, which will have the same performance for both systems, results in the two system having a much smaller performance difference. Our experiments are taking place on a relatively fast, high-bandwidth, low-latency university network: on slower networks, the two systems will have a more similar performance. When we look at the experimental results in this section and Section 6.8.1, we see that while there is a significant performance difference in terms of memory copying when adding transformation modules, it is to a large extent over-shadowed by the performance of the network.

Table 6.9. The average single machine end-to-end time in microsends under shared memory. Columns are grouped by the size of the character array. Rows are grouped by the number of transformation modules.

Size	100	500	1000	5000	10000	50000	100000	500000	1024000
0 TM	41	35	43	42	56	99	137	564	2233
1 TM	63	59	63	75	80	128	154	612	2234
2 TM	79	83	75	87	103	134	207	593	2237
3 TM	104	99	101	99	127	156	180	641	2283
4 TM	118	113	130	125	134	181	207	619	2188
5 TM	126	144	130	136	164	192	233	712	2193
6 TM	154	155	171	161	171	212	253	665	2260
7 TM	173	173	172	175	194	223	262	643	2340
8 TM	195	188	185	206	218	261	343	633	2279

6.9 Video Driver Performance with Added Transformation Modules

We wanted to test the performance of a real networked device driver, using a real device, with both implementations, over varying numbers of transformation modules. For these tests, we used the video driver, which is challenging as it produces a large amount of data relatively quickly. Measuring how our system performs using a real device gave us a realistic idea of its performance, and how much that performance is affected by our different implementations.

6.9.1 Video Inter-frame Times Across the System

We first measured how well the networked device driver as an entire system performs, to give us an idea of overall performance, and how our different implementations effect it. We measured performance in both implementations across different numbers of transformation modules, in order to see how much the addition of a transformation module effects the system performance with a real device.

Table 6.10. The average inter-frame time in microsends under pipes. Columns are grouped by the width/height of the frame - all frames are square. Rows are grouped by the number of transformation modules.

Frame Size	100	200	300	400	500	600	700
0 Trans Mods	2915	11281	27523	43761	75776	112462	165544
1 Trans Mods	3321	12492	30766	49554	84825	112714	166403
2 Trans Mods	3362	12554	31057	51241	86899	115094	169070
3 Trans Mods	3478	12795	31710	51958	88636	117945	172447
4 Trans Mods	3555	12903	32237	53069	91230	120553	176393
5 Trans Mods	3559	13239	33684	54745	92584	124439	179495
6 Trans Mods	3552	13479	34438	55928	94546	125136	182701
7 Trans Mods	3646	13973	34774	57019	96342	126648	185991
8 Trans Mods	3691	14338	35657	58412	98516	128457	189348

Table 6.11. The average inter-frame time in microsends with shared memory. Columns are grouped by the width/height of the frame - all frames are square. Rows are grouped by the number of transformation modules.

Frame Size	100	200	300	400	500	600	700
0 Trans Mods	2576	10297	23227	47972	75864	109041	148327
1 Trans Mods	2876	11664	26845	48269	75334	108052	147537
2 Trans Mods	3088	11680	26506	47000	75642	109044	148317
3 Trans Mods	2961	11985	27138	48199	75689	108325	148042
4 Trans Mods	2949	11924	27277	48398	75235	108492	148277
5 Trans Mods	2952	11871	27040	48081	75519	109543	147986
6 Trans Mods	3060	11915	27019	48025	75343	108811	147808
7 Trans Mods	3064	12068	27137	48367	75887	109354	148276
8 Trans Mods	2930	11867	27014	47847	75152	108769	148178
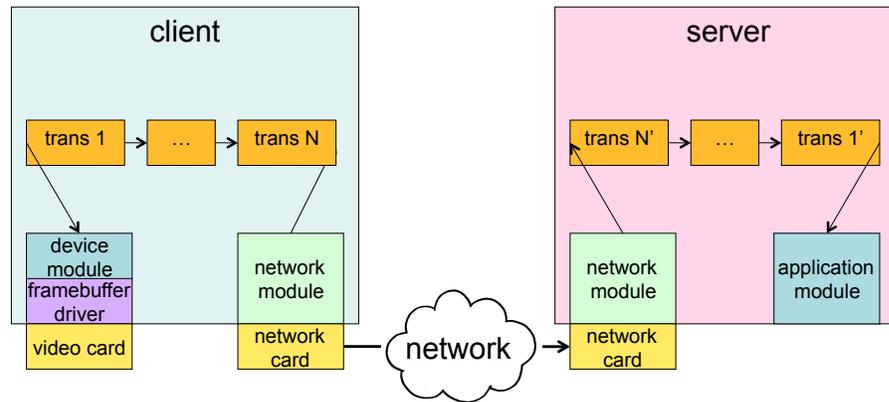
Figure 6.21. The design of the experiment, measuring interframe time across machines. Since we are measuring the time between frames, it is not necessary for the application communication module and device communication module to be on the same machine.

Since we were using video, a natural metric is how many frames per second our system can display. Using this metric let us take time measurements on the same machine, avoiding having to double the system as we did in our experiments in Section 6.1.

To test the overhead of adding additional modules using pipes, a transformation module was created that simply read from an input pipe, and wrote to an output pipe, performing no computation on the data (allowing us to focus purely on communication overhead). The video card network device driver was used to send various sized frames of video through a range of multiple copies of the simple transformation module. The application communication module of the video card was instrumented to measure how long it took to receive and display a frame. Since the application communication module for the video card runs in an infinite loop, reading in a frame from the read pipe and then writing it to the frame buffer, a timestamp was taken at the end of this loop. The previous timestamp was subtracted from the current timestamp to get the inter-frame time, and then average this over five thousand runs for each frame size. This experimental setup is illustrated in Figure 6.21.
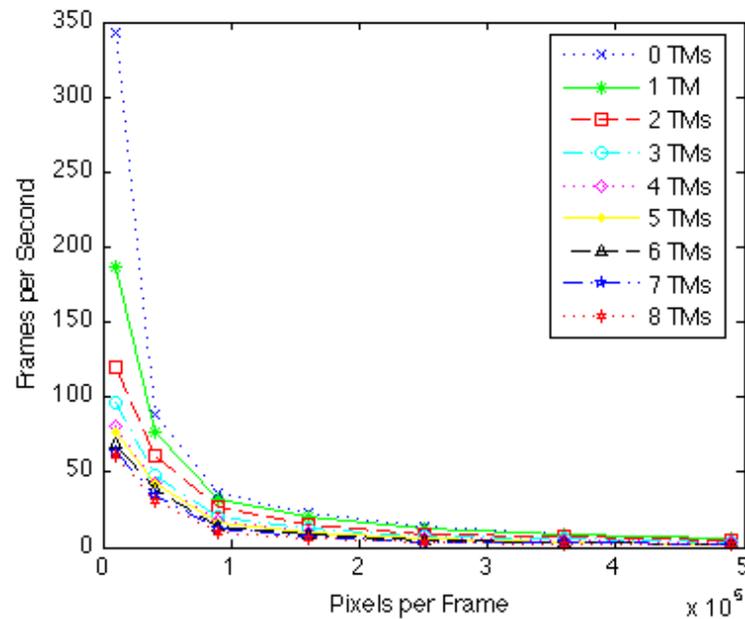
Figure 6.22. The frame rate in frames per second of different sized frames with added transformation modules, using pipes. An additional transformation modules adds approximately two percent of overhead.

To test shared memory, a similar transformation module was created, which simply passed the data through without modifying it. Using our shared memory implementation, this means the transformation module read the starting point and size of the update in shared memory, and then wrote it to the next transformation module, without touching or accessing any of the data in the shared memory.

We first looked at the performance of the pipe-based implementation. As shown in Figures 6.22 and 6.24, adding transformation modules increases overhead, resulting in it taking longer to send and display a frame, as expected. The number of transformation modules listed in the figure is the number of transformation module pairs: for two transformation modules, there are two parts on each side of the network. Even without transformation modules, the data must be moved between the communication module and the network module on each side.
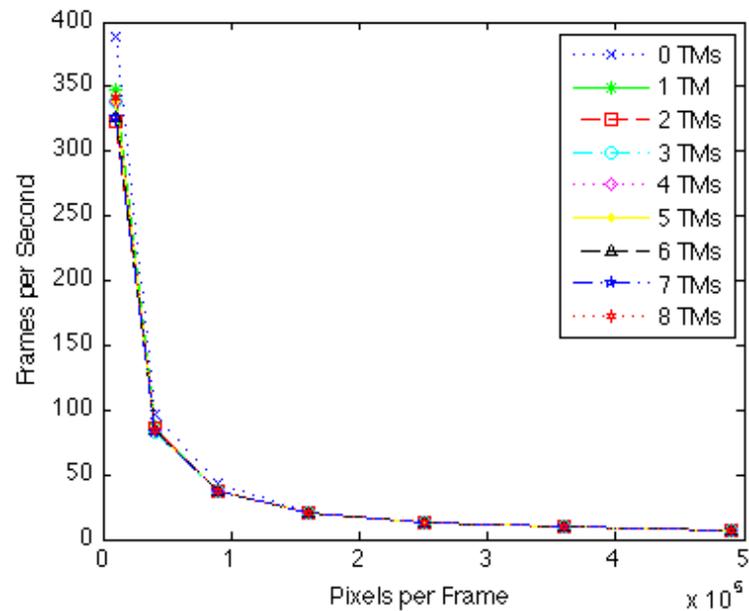
Figure 6.23. The frame rate in frames per second of different sized frames with added transformation modules, using shared memory. Each added transformation module adds 0.36 percent of overhead.

Larger frame sizes are more affected by adding transformation modules, since larger frames result in more data to copy. We tested frame sizes from 100 by 100 to 700 by 700 pixels. With zero transformation modules, it takes 2915 usec between frames for a 100 pixel square frame, and 165544 usec for a 700 pixel square frame, while with 8 transformation modules, it takes 3691 usec for a 100 pixel square frame, and 189348 usec for a 700 pixel square frame. The main performance dip is in adding the first transformation module: this adds 8.91 percent of overhead to the system with no transformation modules. Each additional transformation module adds an average of 2.27 percent overhead.

These incremental overheads are small and tolerable, and in a sense are indicative of worst-case cost since they will only become a smaller portion of the overall time when the transformation modules actually do useful work (and thus take up time themselves), or when network times grow beyond that of the fast network used in our experiments.
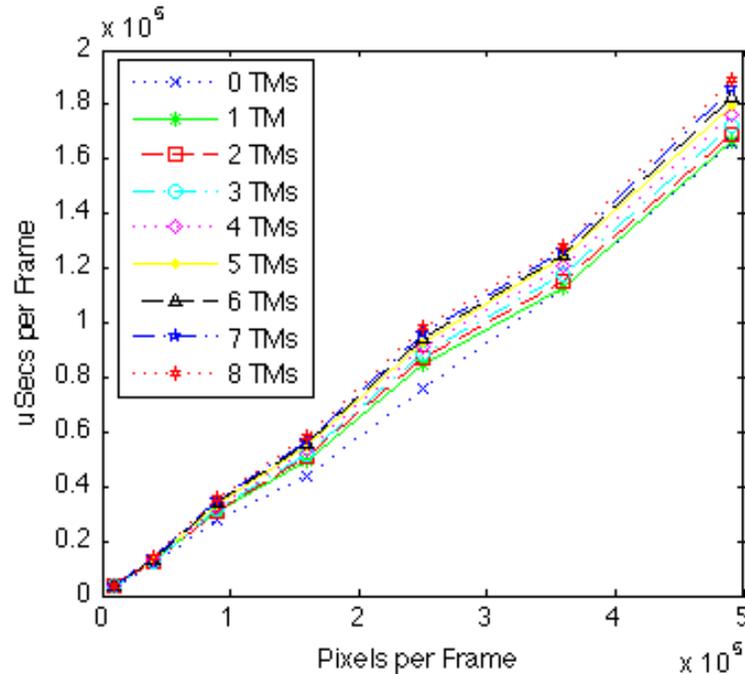
Figure 6.24. The average inter-frame time in microseconds per frame of different sized frames with added transformation modules, using pipes. As the framesize increases, so does the impact of adding an additional transformation module.

As show in Figures 6.23 and 6.25, the times in the shared memory implementation are essentially unaffected by adding transformation modules. Without transformation modules, it took the system 2576 usecs to send a 100 by 100 pixel frame, and 148327 usecs to send a 700 by 700 pixel frame. With eight transformation modules, it took 2930 usecs for a 100 by 100 pixel frame, and 148178 usecs for a 700 by 700 pixel frame. Again, the biggest performance dip is the first transformation module, which added 5.57 percent of overhead to the system. Each additional transformation module added an average of 0.36 percent.

The shared memory implementation added very little overhead with each additional transformation module. This makes it very well suited for devices such as the video card, which produce a large amount of data.

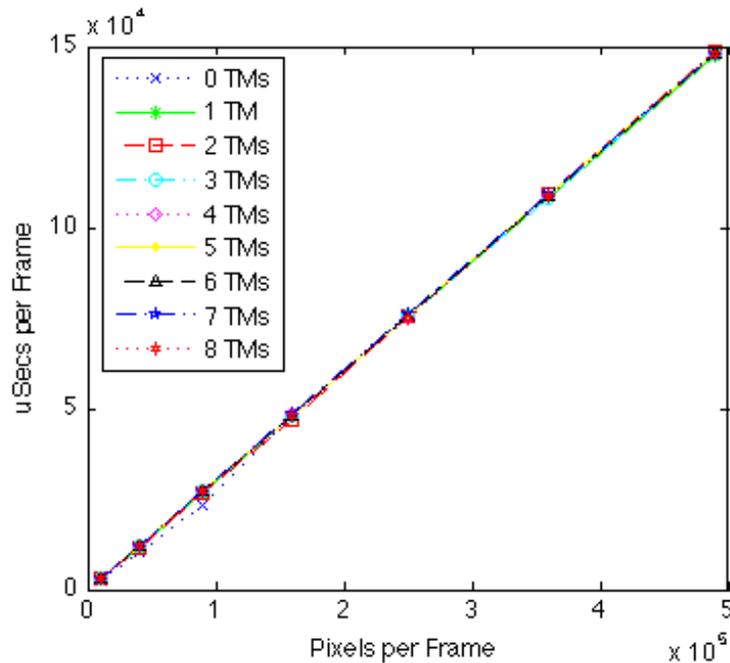In Figure 6.26 we showed the average time it takes between transfers of a frame

Figure 6.25. The average inter-frame time in microseconds per frame of different sized frames with added transformation modules, using shared memory. Performance remains relatively constant across the number of transformation modules.

of video in both shared memory and pipe implementations, across varying numbers of transformation modules. Focusing on the minimal times (with no transformation modules), for a 100 square pixel frame, it took 2915 usecs using pipes and 2576 usecs using shared memory. For a 700 square pixel frame, it took 165544 usecs using pipes and 148327 usecs using shared memory. From the rest of the graph, one can see that additional overhead from using pipes grew with both the number of transformation modules and the frame size, though the slopes are not large.

Without transformation modules, the system using pipes took 6.72 percent longer to display a frame on average. Each additional transformation module added 2.34 percent of overhead when compared to the shared memory implementation using the same number of transformation modules. Again, this difference in overhead is a worst-case scenario since the transformation modules were simply transferring data. In more realistic
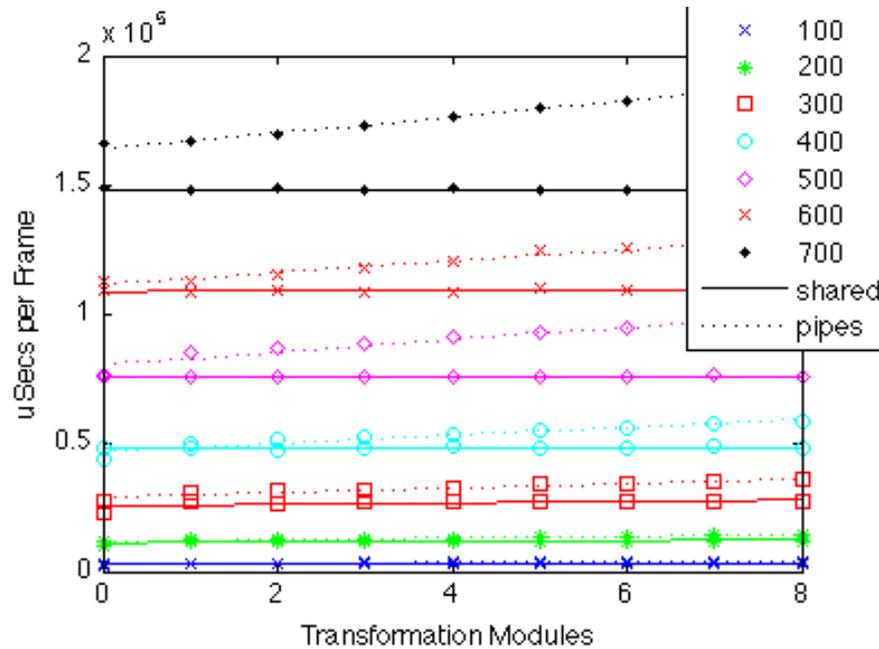
Figure 6.26. Performance of pipes versus that of shared memory, grouped by size of frame. Additional transformation modules increase the performance difference between pipes and shared memory by approximately two percent.

cases where the modules are doing a computation, the differences in transfer speed will be a smaller proportion of the processing time, and slower network transfer times will also cause the intra-machine transfer time to have less impact. As in our results from Section 6.8.1, we see that with large amounts of data and many transformation modules, shared memory provides us with a significant performance advantage. For devices with smaller amounts of data, or with few transformation modules, the two implementations perform similarly.

6.9.2   Single Machine End-to-End Time and Batching Effects

We next looked at the results from a single machine, which gave us a view of performance without the effects of the network, making the differences in performance for our different implementations clearer.

We measured the end-to-end time on a single machine, comparing over different
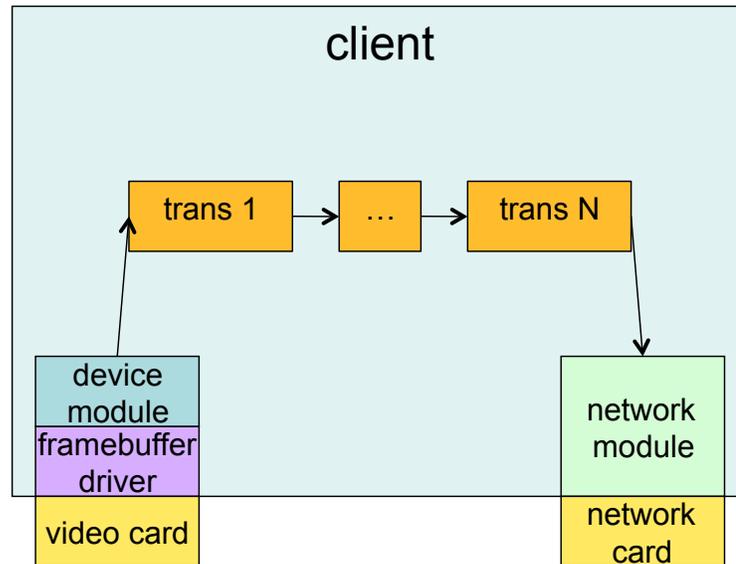
Figure 6.27. The design of the experiment, measuring end-to-end time for one machine. A message is timestamped right after creation in the device communication module, and right before being sent to the network by the network module, and the difference is taken.

numbers of transformation modules. Note that while in our experiments in Section 6.9.1 were measuring the time in between frames, we were now measuring the time a single frame takes from being copied into the device communication module until it is ready to be sent over the network by the network module. Our experimental design, illustrated in Figure 6.27, was identical to our experiments in Section 6.2.

Our results are shown in Figure 6.28 and Tables 6.12 and 6.13. The most startling characteristic of these results is that the pipes have shorter end-to-end times than the shared memory implementation, especially at large frame sizes. For a frame of 700 square pixels, with 8 transformation modules, we found an end-to-end time of 211646 usec using pipes, and an end-to-end time of 249453 usec using shared memory. This is counter to all of our previous results, in which shared memory performed better, especially with larger messages. This is especially mysterious since the inter-frame time for the video card driver does not display this same pattern. As shown in our results from Section

Table 6.12. The average end-to-end time in microsends with pipes, on a single machine. Columns are grouped by the width/height of the frame - all frames are square. Rows are grouped by the number of transformation modules. All measurements are taken in single-core mode.

Frame Size	100	200	300	400	500	600	700
0 Trans Mods	3420	12353	32267	52653	86687	117514	175065
1 Trans Mods	4030	13399	32684	53141	90701	119323	177204
2 Trans Mods	3746	13686	33138	54289	91995	122109	182296
3 Trans Mods	4234	13705	33516	54713	94095	124431	185870
4 Trans Mods	4387	13944	34042	56571	95657	127190	189502
5 Trans Mods	4208	14118	34515	57399	98632	131760	194653
6 Trans Mods	4581	14620	34951	59234	100573	135676	199830
7 Trans Mods	4652	15037	35729	59906	103035	139448	205886
8 Trans Mods	4599	15120	36186	61878	105466	142826	211646

Table 6.13. The average end-to-end time in microsends with shared memory, on a single machine. Columns are grouped by the width/height of the frame - all frames are square. Rows are grouped by the number of transformation modules. All measurements are taken in single-core mode.

| Frame Size	100	200	300	400	500	600	700
0 Trans Mods	3457	14539	36462	73862	127105	182598	249516
1 Trans Mods	3453	14604	36341	73995	126938	182240	249465
2 Trans Mods	3354	14294	35958	73397	126939	182196	249465
3 Trans Mods	3351	14320	35956	73345	126940	182200	249412
4 Trans Mods	3354	14284	35880	73323	126875	182153	249452
5 Trans Mods	3284	14232	35923	73474	126895	182186	249424
6 Trans Mods	3384	14106	35834	73369	126951	182197	249476
7 Trans Mods	3352	14302	35985	73280	127001	182235	249393
8 Trans Mods	3366	14233	35891	73357	126961	182161	249453
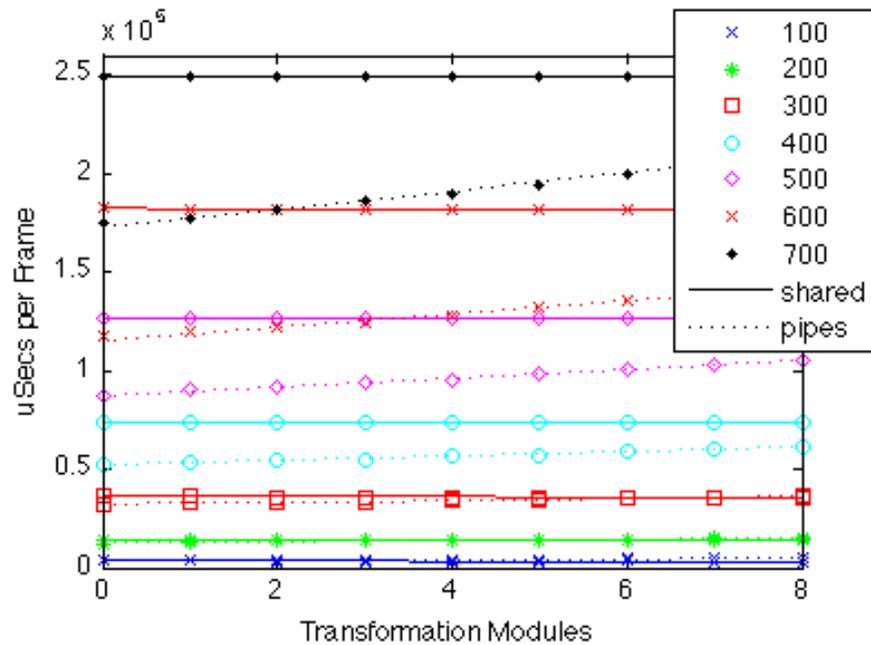
Figure 6.28. The end-to-end time on one machine, comparing pipes and shared memory. On a single machine, shared memory actually has higher end-to-end times than pipes for video, especially for large frames.

6.9.1, the inter-frame times are lower for the shared memory implementation.

We hypothesized that the larger end-to-end times for shared memory were due to batching in the system, i.e. each module processing several frames at a time, rather than processing a single frame and then switching to a new process. In our experiments, three "slots" for frames were open in shared memory, allowing modules to process up to three frames at a time, if they were available. This batching would cause the average end-to-end to be greater, as the frames would have to wait until the entire batch is processed before being moved to the next module. This batching effect did not appear in pipes, since the system was writing a large amount of data to the pipe. This caused the write to block when the pipe is full, forcing a switch to one of the waiting processes, which would in turn read from the pipe and process the message.

In order to clarify, we provide an example with, hypothetical times:

Say each process takes 2 usec to process, and 3 usec to switch between processes,
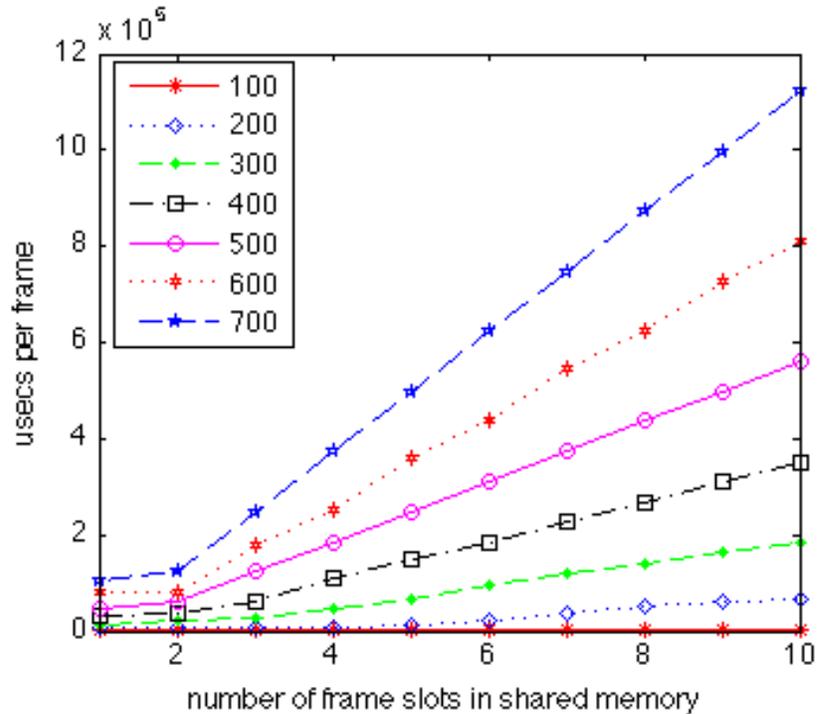
Figure 6.29. The end-to-end time on one machine, in microseconds, with differing numbers of frames in shared memory, grouped by size of frame. The more frames are available in shared memory, the higher the end-to-end time.

and there are 3 processes.

Without batching (i.e. for pipes), the end-to-end time for each update will be 2 usec + 3 usec + 2usec + 3 usec + 2 usec, or 12 usec.

Now, let's say this is happening in batches of 10 updates. For the first update, its end-to-end time will be 20 usec + 3 usec + 20 usec + 3 usec + 2 usec, or 48 usec, since it has to wait for all 10 updates to be processed in all but the last update. The last update in the batch will be 2 usec + 3 usec + 2 usec + 3 usec + 2 usec, or 12 usec. So the average end-to-end time will be 10 usec + 3 usec + 10 usec + 3 usec + 2 usec, or 28 usec, much larger than the non-batched version.

However, the inter-frame time will be larger without batching: without pipelining effects, we would have 12 usec inter-frame without batching, versus an average of (20

Table 6.14. The average end-to-end time in microsends with shared memory. Columns are grouped by the width/height of the frame - all frames are square. Rows are grouped by the number of frames stored in shared memory at once. All measurements are taken in single-core mode, with no transformation modules

Frame Size	100	200	300	400	500	600	700
1 Frame Slot	1206	4936	14041	32307	45471	78654	106673
2 Frame Slots	1210	7064	21545	37435	61052	82642	123043
3 Frame Slots	1213	8383	24550	61004	122869	178087	248269
4 Frame Slots	1216	9201	46222	110494	186068	254234	372837
5 Frame Slots	1219	11522	66958	147838	248755	360942	498348
6 Frame Slots	1226	21258	97038	184979	311316	440049	623286
7 Frame Slots	1241	38465	118381	228818	374561	543835	748093
8 Frame Slots	1255	50711	141260	268043	437147	625666	873701
9 Frame Slots	2021	59535	164374	309114	499082	727716	998573
10 Frame Slots	1572	68111	183734	352250	562398	812175	1123959

usec + 3 usec + 20 usec +3 usec + 20 usec)/10 = 6.6 usec for a batch of 10.

In order to support our batching hypothesis, we performed an experiment in which we measured the end-to-end time of two modules with no transformation modules, keeping a varying number of frames in shared memory. As shown in Figure 6.29 and Table 6.14, the end-to-end time increases as more frames are available in shared memory. This indicates that batching is what is causing end-to-end times to be longer for shared memory than for pipes.

This result has important ramifications for the design of shared-memory-based networked device drivers. Designers must decide how many frame "slots" they wish to make available in the pool of shared memory. More available slots will result in a more parallelization within the system, supporting shorter inter-frame times, but also more batching, resulting in longer end-to-end times and more jitter.

6.10   Conclusion

Throughout this chapter, we compare performance results two implementations, one using pipes, and one using shared memory. We see that thanks to its avoidance of memory copies between processes, the shared memory implementation has less performance overhead, especially for large updates, or large numbers of transformation modules. However, we also show that the performance of our pipe-based implementation is not overwhelmingly high, especially for devices with smaller updates, or networked device drivers with fewer transformation modules.

We believe that pipes provide a significant ease of implementation that justifies their higher performance overhead. As described in Section 5.1.3, shared memory requires a great deal of additional structure when designing and implementing modules. We see this in our results from Section 6.9.2, which show that designers must consider several trade-offs when choosing how many frames to hold in shared memory. When using shared memory, designers must also consider how they will handle reserving and freeing memory, what to do about messages that change size throughout the system, and a number of other factors that are already taken care of when using pipes. Because of this, the higher overhead of pipes becomes a worthwhile trade-off for many networked device drivers.

Chapter 6, in part, has been submitted for publication of the material as "Performance Aspects of Data Transfer in a New Networked I/O Architecture by Cynthia Taylor and Joseph Pasquale. The dissertation author was the primary investigator and author of this paper.

Chapter 6, in part, is a reprint of the material as it appears in the article "A Remote

I/O Solution for the Cloud", by Cynthia Taylor and Joseph Pasquale, which appears in

the Proceedings of the 5th International Conference on Cloud Computing, Honolulu, HI,

June 2012. The dissertation author was the primary investigator and author of this paper.

Chapter 7

Conclusion

In this work, we presented a new networked device driver architecture to support remote I/O devices. We support network transparency while also allowing processing of data in order to compensate for the network. Our system is easy to extend to new devices, and supports customization and the addition of new features.

To support network transparency, we encapsulated all networking and related processing inside the networked device driver. This means applications do not need to be modified to work with remote devices, since the network is in effect invisible to both application and device. This makes it easy to support legacy applications, avoids parallel code maintenance, and means that a developer can develop a network device driver for an application, without having the source code for the application.

Inserting the network between the application and device means our system needs to be able to handle issues such as security, latency, bandwidth restrictions, and jitter. Furthermore, each device and application may have a different ideal solution to these problems, requiring different solutions for different device and application pairings. We introduced the idea of transformation modules, processes which enact transformations on the I/O stream. These modules are invisible to the application and device, yet let the system be customized for the exact needs of an individual application and device, and their particular network conditions.

We introduced transformation module pairs, a key concept of our system. Each transformation module on the client is paired with a matching transformation module on the server. The client transformation module performs some transformation on the data stream, and the server transformation module reverses this transformation, e.g. compressing and decompressing. This pairing concept also allows us to automatically order the transformation modules, since operations must be undone in the reverse order of how they were applied (e.g. if the data stream is encrypted and then compressed, it must be decompressed and then decrypted).

The network device driver architecture is designed to run within the application layer whenever possible. Our system run within the kernel only when necessary to preserve transparency. This avoids the security dangers that come with allowing arbitrary code to run in the kernel. It also allows to leverage many existing mechanisms supplied by the kernel. The main unit of modularity within our system is the process. Separate functional modules each run as their own process. This allows us to use the kernel's scheduler, rather than having to write our own.

We presented two separate implementations of our system, one which uses pipes to transfer data between modules, and one which uses shared memory. We showed that the implementation using pipes incurs an overhead that, while expectedly more overhead than that of the shared memory implementation, is relatively small and tolerable. The benefit of pipes, of course, is their ease of use and the simplicity that results in the system's implementation. Using pipes are appropriate for many applications, especially those that leverage their built-in flexibility when processing variably-sized messages

In our performance evaluation, we showed that our system adds a relatively small amount of overhead. We compared a networked device driver for the video card with VNC, a popular thin client software designed to efficiently transfer video, and show that the networked device driver has similar performance. We also showed that using

transformation modules can result in a significant performance advantage over sending raw data over the network.

In conclusion, we presented the networked device driver architecture, a distributed architecture for remote I/O. This architecture supports network transparency, while at the same time allowing processing of the data stream for the network. It provides a modular, flexible solution which is easy to extend for new devices and new functionality.

Bibliography

[1] Tight VNC. URL http://www.tightvnc.com/.

[2] 3d Connexion. 3d Connexion Space Navigator. URL http://www.3dconnexion.com/.

[3] R.A. Baratto, L.N. Kim, and J. Nieh. Thinc: a virtual display architecture for thin-client computing. In *ACM SIGOPS Operating Systems Review*, volume 39, pages 277–290. ACM, 2005.

[4] T. Hirofuchi, E. Kawai, K. Fujikawa, and H. Sunahara. USB/IP: a peripheral bus extension for device sharing over IP network. In *Proceedings of the annual conference on USENIX Annual Technical Conference*, page 42. USENIX Association, 2005.

[5] J. Huang, W. Feng, N. Bulusu, and W. Feng. Cascades: Scalable, flexible and composable middleware for multimodal sensor networking applications. In *Proceedings of The ACM/SPIE Multimedia Computing and Networking*. Citeseer, 2006.

[6] T.C. Hudson, A. Seeger, H. Weber, J. Juliano, and A.T. Helser. VRPN: a device-independent, network-transparent VR peripheral system. In *Proceedings of the ACM symposium on Virtual reality software and technology*, pages 55–61. ACM New York, NY, USA, 2001.

[7] E. Kohler, R. Morris, B. Chen, J. Jannotti, and M.F. Kaashoek. The click modular router. *ACM Transactions on Computer Systems (TOCS)*, 18(3):263–297, 2000.

[8] J. Kong, I. Ganev, K. Schwan, and P. Widener. Cameracast: Flexible access to remote video sensors. In *Proceedings of the ACM Multimedia Computing and Networking Conference (MMCN)*. Citeseer, 2007.

[9] K. Mayer-Patel and L.A. Rowe. Design and performance of the berkeley continuous media toolkit. In *Multimedia Computing and Networking*, pages 194–206. Citeseer, 1997.

[10] X. Meng, J. Shi, X. Liu, H. Liu, and L. Wang. Legacy application migration to cloud. In *Cloud Computing (CLOUD), 2011 IEEE International Conference on*, pages 750–751. IEEE, 2011.

[11] J. Pasquale, E. Anderson, and P.K. Muller. Container shipping: operating system support for i/o-intensive applications. *Computer*, 27(3):84–93, 1994.

[12] C. Phanouriou and M. Abrams. Transforming command-line driven systems to web applications. *Computer Networks and ISDN Systems*, 29(8-13):1497–1505, 1997.

[13] R. Pike. $8\frac{1}{2}$, the Plan 9 window system. In *Proceedings of the Summer 1991 USENIX Conference, Nashville, TN, USA, June 10–14, 1991*, pages 257–265 (of x 473), Berkeley, CA, USA, 1991. USENIX. URL citeseer.ist.psu.edu/article/pike91plan.html.

[14] D. Presotto, R. Pike, K. Thompson, and H. Trickey. Plan 9: A distributed system. In *Proceedings of Spring 1991 EurOpen*, pages 49–56, 1991. URL citeseer.ist.psu.edu/presotto91plan.html.

[15] R. Recio, P. Culley, D. Garcia, J. Hilland, and B. Metzler. An rdma protocol specification. Technical report, IETF Internet-draft draft-ietf-rddp-rdmap-03. txt (work in progress), 2005.

[16] G. Reitmayr and D. Schmalstieg. OpenTracker: A flexible software design for three-dimensional interaction. *Virtual Reality*, 9(1):79–92, 2005.

[17] T. Richardson, Q. Stafford-Fraser, K.R. Wood, and A Hopper. Virtual network computing. *Internet Computing*, 2(1):33–38, 1998.

[18] Tristan Richardson. The RFB Protocol. Technical report, RealVNC Ltd, 2007.

[19] D.M. Ritchie. A stream input-output system. *AT&T Bell Laboratories Technical Journal*, 63(8):1897–1910, 1984.

[20] Robert W. Scheifler and Jim Gettys. The x window system. *ACM Trans. Graph.*, 5 (2):79–109, 1986. ISSN 0730-0301. doi: http://doi.acm.org/10.1145/22949.24053.

[21] S. Venkateswaran. *Essential Linux device drivers*. Prentice Hall Press, 2008.

[22] J. von Spiczak, E. Samset, S. DiMaio, G. Reitmayr, D. Schmalstieg, C. Burghart, and R. Kikinis. Multi-modal event streams for virtual reality.. In *Proc. 14th SPIE Annual Multimedia Computing and Networking Conference (MMCN'07), San Jose,*

California, 2007.

[23] C. Waldspurger and M. Rosenblum. I/o virtualization. *Communications of the ACM*, 55(1):66–73, 2012.