A Highly-Extensible Architecture for Networked I/O

Cynthia Taylor
Computer Science
Oberlin College
Oberlin, OH
Cynthia.Taylor@oberlin.edu

Joseph Pasquale
Computer Science and Engineering
University of California, San Diego
La Jolla, CA
pasquale@cs.ucsd.edu

***Abstract*—We describe a new distributed I/O software architecture to support remote applications interacting with local I/O devices based on the concept of a networked device driver. Our goal is both network transparency and high extensibility/ease of customization in support of the vastly different types of applications and devices that can benefit from remote I/O, especially relevant in cloud computing contexts. A networked device driver logically connects a device at one network endpoint and an application at another, and allows the I/O stream between them to be modified by a set of pipelined transformation modules. Each transformation module comes in a pair, operating on each side of the network, with one side typically applying some operation and the other side applying a corresponding one (e.g., one that reverses the original transformation). Because of the paired nature of transformation modules, the system is capable of supporting the modification of the I/O stream in a variety of ways to compensate for network issues while remaining transparent to the application, and also results in a high degree of extensibility. This is achieved with a mostly user-level implementation that incurs a relatively low degree of overhead.**

I. Introduction

In this work, we present the design of a system software architecture where devices can communicate over the network with applications transparently, without applications having to be specifically designed for networked use. Legacy applications that were designed to run with devices locally should be able to run on this system with no modification. The 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.

A motivation for remote I/O in general is the ability to integrate computation-and-data intensive applications with lightweight devices. A typical approach to this is thin client computing, where the application is moved from the local machine to a more powerful machine, and all user I/O is forwarded. A drawback to traditional thin clients is that they were not designed to support the current wide array of I/O devices, typically limiting their support to the mouse, keyboard and video card.

In this work, we present a general system which can be easily extended to any device. Given the plethora of new devices being developed, and the growing number of applications for which it is desired to use them, a general system for supporting remote I/O is very relevant, especially in cloud computing contexts that are becoming more and more prevalent. The ability to quickly integrate new devices, as well as the ability to flexibly modify an I/O stream that may be highly specific to a particular device and application, are key capabilities of our system.

For example, one of the chief issues of remote I/O is the disruption to the flow of messages between device and application caused by the network. In addition to performance-related issues, the network may introduce security issues as well. While previous and long-standing work on network protocols have addressed these issues at the network/system level, it is generally difficult for the user to configure a protocol that is tailor-made for the application and device(s) they are about to invoke, and one that may take into account existing or predicted network conditions. Consequently, depending on the specific device, application, and qualities of the network, the ideal matter of compensating for these issues will vary. To address this, our architecture supports the creation of customizable transformation modules, designed to dynamically process device or application information.

In this paper, we describe the System Architecture, Implementation, and selected Performance Results. For more detail, including more experimental results, please see [1].

II. System Architecture

A. Architecture Summary

The system architecture is best understood as a series of modules, each of which can be created separately and combined with existing modules to form a *networked device driver*. The networked device driver spans both local and remote machines, half operating on one and half on the other, as shown in Figure 1. The resulting networked device driver must be able to support message passing via the network, between a device and application which reside on separate machines. To support network transparency, all network-related computation is contained within this driver. Messages, created by either device or application, are transformed by one side of the networked device driver, and passed over the network. The other half of the networked device driver reads these messages from the network, reverses the transformation to preserve transparency, and forwards them to their destination.

A networked device driver manages an I/O stream, i.e., a sequence of messages, traveling in a single direction, either from device to application, or from application to device. The messages are created, and retrieved from their source by either the *device communication module* or the *application communication module*. They then go through an optional series of *transformation modules*, each of which changes the messages in some fashion. The *network modules* then pass the messages over the network. They then travel through a matching series of transformation modules on the destination machine, which reverse any changes the original transformation modules made. Lastly, they are sent to either the device communication module or application communication module, which transmits them to their final destination.

The transformation modules allow messages to be modified to be more effectively transmitted over the network, only to then be returned to their original format to be passed to their final destination so that the network seems transparent. In addition, having separate and possibly asymmetric networked device drivers for application-to-device and device-to-application communication allows our system to handle each communication stream in a way that best fits its unique data characteristics.

B. Implementation

We designed the system so that it is simple to create and customize networked device drivers. It is also designed to be easily extensible, so that device manufacturers, application designers and end users can all build system modules that meet their unique needs. With this in mind, we implemented the system predominantly at the user level (i.e., outside the operating system). This avoids the dangers of running arbitrary code in the kernel, both in terms of possible malicious access, and system failures on errors. Our system uses the existing kernel scheduler, with each module running as a separate user-level process. This also allows processes to automatically block when there is no more data for them to process. All modules except the application communication module run exclusively at user level.

To create a networked device driver for a new device, a designer needs to provide a device communication module that uses the API for the device, and an application communication module which follows the interface of the original driver for the device. These new modules can be combined with any existing transformation and network modules to form a new networked device driver.

C. Device Communication Module

The device communication module is simply a user-level application which interacts with the device using its API. It receives information from the original, unmodified device driver supplied by the device manufacturers or operating system. Each device must have its own custom device communication module, since the module communicates with the

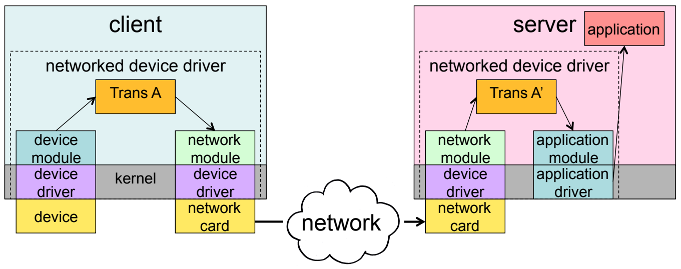

Fig. 1. The software architecture of a networked device driver. Note that only the application driver and raw driver run within the kernel.

device using its specific API. However, the generality of the device communication module may vary: for example, we wrote a generic mouse device communication module which uses generic linux mouse events, but a device communication module for a specific mouse could be written on a lower level.

D. Network Modules

There are a pair of network modules, one on each machine. Their job is to send data over the network; all other functionality, no matter how closely related to the network protocol, is done by the transformation modules. The processing the network modules do is to send an entire update at a time, rather than transmitting partial updates as they are received.

We have implemented network modules using the TCP protocol. For applications and devices where speed is more important than ordering or consistency, UDP may be a natural fit, and could easily be implemented. It would also be easy to create modules for new or experimental network protocols.

E. Application Communication Module

The application communication module communicates with the application using the interface of the original device driver for the device. Each device must have its own, custom-written application communication module, since the module imitates the original driver for the device, to provide transparency. Because of this, the application communication module is the one module which may run within the kernel. If the original device driver runs entirely within the kernel, the application communication module will have two pieces, one of which runs at user-level, and one which runs within the kernel. This kernel component is required for the application to be able to use the driver without modification.

III. Transformation Modules

Messages pass through a set of optional, paired transformation modules before they reach their destination. These modules add extra functionality to the system, while their pairing allows us to keep transparency. Each module performs a specific task, providing extra functionality for the system. What extra functionality is required depends on the device, the needs of the application, and the characteristics of the network. Some example functions include averaging multiple updates together, buffering, compression, encryption, and synchronizing multiple devices.

The level of device specificity will vary with the functionality. For example, averaging messages together requires semantic knowledge about the message format, and so averaging modules must be designed for a specific device. In contrast, functions such as encryption do not need to know anything about message format, and so can be used for any device. Transformation modules for specific devices can easily be combined with generic transformation modules, making it simple to leverage existing modules for a customized network device driver.

Any changes made to a message must eventually be reversed, to preserve transparency. For example, if a message is compressed before it is sent over the network, it must be decompressed before it reaches the application. These reversals must be performed in the opposite order of the original operations: if a message was compressed and then encrypted, it must be decrypted and then decompressed. We define transformation module pairs, one of which performs an action on the client, and the other reversing the action on the server. These pairs are automatically arranged so that their order on the server is the reverse of their order on the client.

Each message passed between the modules has a header specifying the length of the message, and any optional information about the message needed by the transformation module's pair module to reverse the transformation. This extra header information is stripped off by the module on the server, so that the message reaches further modules in the state that they expect.

A pair of transformation modules may communicate directly with each other to exchange control information about data processing. We support this out-of-band communication, as the messages between the transformation modules will have different data characteristics than the messages between the device and application. This allows transformation modules to change how they are processing data as a pair, to respond to changes in network conditions.

IV. Performance

In this section, we present performance results of various tests we ran on different aspects of our implementation. The purpose of these tests was to ensure that our approach to implementation (separating modules into different processes, working at the user-level, etc) was not causing overly high overhead. With this in mind, we test the effects of adding transformation modules specifically, as well as testing the system as a whole.

To measure performance, we ran experiments on two computers, both off-the-shelf Dell PCs. We used a Dell Optiplex 755 with an Intel Core 2 Duo chip and a 333 MHz FSB clock, and a Dell Optiplex 320 with an Intel Celeron Chip and a 133 MHz FSB clock. We used these machines because they are both examples of relatively inexpensive off-the-shelf hardware that would be available to anyone. Both machines are running Ubuntu Linux. Both machines have wired connections to a relatively fast campus network, with a sample ping round-trip time of 0.235 msec.

TABLE I
The average time to for a message, consisting of a random character array, to traverse the system, in microseconds.

bytes	1000	5000	10000	100000	1024000
usec	632	819	1287	10320	91988

A. Base Speed of Drivers

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.

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. 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 clocks across machines with enough precision to make our measurements. 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. To get the end-to-end times from the round trip times being measured, the times were divided by two. All measurements are averaged over 450 tests, as this gave us sufficient data points while also allowing the tests to complete in a reasonable time.

Our results are shown in Table I. We demonstrated that our system incurs a relatively small amount of overhead, especially for smaller message sizes. For example, 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.

B. Buffering

To demonstrate our buffer modules' ability to recreate the timing of updates across the network, we created two network device drivers, one with the buffer modules, and one without.

We used a version of the spacenav driver that automatically generates updates every 16 msec. These updates consist of a time stamp and 6 integer values. This was read 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. We set the sink buffer module to have a maximum buffer of 10 updates, and the source side buffer to have a maximum buffer of 400 updates. We measured the inter-message distance in the application communication module. We must measure it here instead of in the application because the application communication module strips off timestamps before passing an update to the application, to preserve transparency.

We compared the results from our buffered network device driver against a network device driver which was set up exactly the same, except without the buffering module. Both network device drivers use TCP-based network modules.

To measure the jitter in both systems, we compared the inter-message time for the updates when they were generated 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 generated a timestamp when it received a new update, used that to compute the time between it and the last update received, and compared it to the time between their generation timestamps. Our jitter metric is 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 msec to 200 msec, created using netem [2]. Netem creates this jitter by delaying sending each packet for a random number of milliseconds between 0 and N, where N is the specified maximum. We then averaged over 200 runs. In Figure 2, we show the average offset time as well as the standard deviation. 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 averaged less than 1.5 msec, and until the added jitter was over 50 msec, it averaged less than .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 was under 1 msec with 50 msec of jitter with buffering, while without buffering it was already at 12 msec.

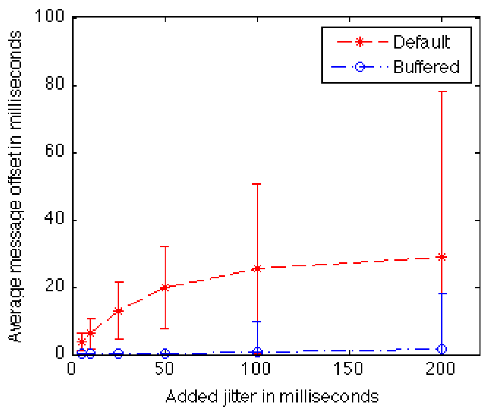

Fig. 2. Average jitter, with standard deviation. Using the buffering module drastically lowers the effects of jitter on the system.

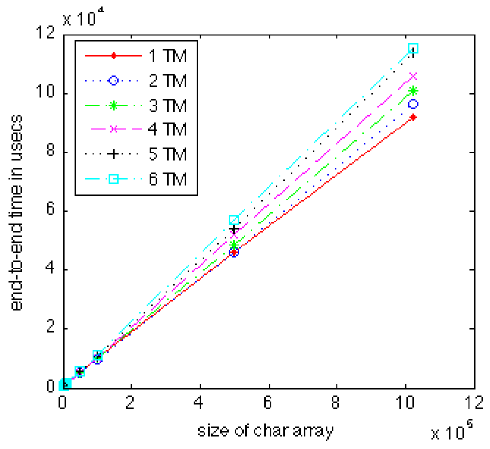

Fig. 3. End-to-end times for pipes, across the entire system, in microseconds. Each transformation module adds an average of 6.6 percent of overhead.

C. Adding Transformation Modules

We 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 how much extra time we add to each message with new transformation. Our experimental setup was the same as in Section IV-A, with the addition of varying numbers of transformation modules. We created transformation modules which simply passed the data to the next module without performing any computation on it, to observe how much overhead was due to the copying itself.

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. It took a 1024000 character array 91988 usec to complete the pipe system with no transformation modules, and 122761 usec with 6 transformation modules.

Each transformation module added an average of 6.6 percent of overhead. 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.

V. Related Work

Two typical thin client techniques are demonstrated by X-Windows and VNC [3], [4]. In X-Windows, high-level commands are captured on the application machine, and forwarded to the device machine. In VNC, pixel data is copied from the framebuffer and sent to the client, where it is displayed by an application. Both are limited in that it they are designed expressly for mouse, keyboard and display. In addition, applications must be written expressly for X. A classic approach to using modules to alter data within a single machine is UNIX Streams, in which a stream between an application and a device passes through modules that alter the data being passed through them [5]. This system runs within the kernel, and is not designed specifically as a distributed system (though one can be built out of them).

USB over IP sends device data at the device controller level, below the individual driver [6]. This allows network transparency and support of any USB device. However, there is no way to allow for specialized treatment of messages from different devices. THINC [7] and CameraCast [8] are both systems designed for video which use logical drivers. These systems are major improvements in dealing with remote I/O in a more abstract way. However, they are both targeted for a specific set of devices, albeit important ones. In addition, THINC does not allow modules to process device data.

Many systems exist that support distributed systems for devices within a specific area, and allow applications to be written for these systems, frequently allowing data from the device to be sent through a series of processing modules. The Berkeley Continuous Media Toolkit does this for multimedia applications, the Cascades system does this for sensor networks, and a large variety of these systems exist within the virtual reality domain [9], [10], [11], [12], [13]. The RAMP system also creates middleware at the application layer for the spontaneous creation of networks which share content and resources [14].

VI. Conclusion

We created the networked device driver architecture to support remote I/O devices in a manner which was both simple and flexible. Network transparency, a major goal, is achieved by encapsulating all networking and related processing of data inside of the device driver. This means that applications will not have to be modified to work with remote devices. Customizability is supported via transformation modules. 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 implemented transformation modules as separate processes that communicate via pipes, making it easy to add and remove transformation modules to drivers to support the needs of different devices and applications. Our user-level approach also makes it simple for developers to create new networked device drivers, as we leverage existing mechanisms as much as possible, while avoiding the potential to introduce dangerous bugs at the kernel level.

Our system's basic abstraction is a one-way communication path between the device and application, generally separated by a network, and which can be modified by paired transformation modules. By designing the system for a device and application at this level, a high degree of customization is possible. Modules can be written for the exact data profile of this communication: the size, frequency, and format of updates can all be considered, both in how they are produced by the source, and what the sink requires from them.

Our performance experiments showed that a user-level implementation relying on standard operating system processes and pipes does not create significant overhead for at least small but expectedly typical numbers of transformation modules. Finally, we demonstrated how adding transformation modules such as buffering can cause a significant gain in performance for certain common and important application scenarios. As future work, we plan to look at both scalability issues, and how the system performs over wireless networks.

References

[1] C. Taylor, "The Networked Device Driver Architecture: A Solution for Remote I/O," Ph.D. dissertation, University of California, San Diego, 2012.

[2] "NetEm." [Online]. Available: http://www.linux-foundation.org/en/Net:Netem

[3] R. W. Scheifler and J. Gettys, "The X Window System," *ACM Trans. Graph.*, vol. 5, no. 2, pp. 79–109, 1986.

[4] T. Richardson, Q. Stafford-Fraser, K. Wood, and A. Hopper, "Virtual Network Computing," *Internet Computing*, vol. 2, no. 1, pp. 33–38, 1998.

[5] D. Ritchie, "A Stream Input-Output System," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910, 1984.

[6] 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*. USENIX Association, 2005, p. 42.

[7] R. A. Baratto, J. Nieh, and L. Kim, "THINC: A Remote Display Architecture for Thin-Client Computing," Department of Computer Science, Columbia University, Computing Science Technical Report CUCS-027-04, 2004.

[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)*, 2007.

[9] K. Mayer-Patel and L. Rowe, "Design and Performance of the Berkeley Continuous Media Toolkit," in *Multimedia Computing and Networking*. Citeseer, 1997, pp. 194–206.

[10] 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.

[11] G. Reitmayr and D. Schmalstieg, "OpenTracker: A Flexible Software Design for Three-Dimensional Interaction," *Virtual Reality*, vol. 9, no. 1, pp. 79–92, 2005.

[12] 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.

[13] T. Hudson, A. Seeger, H. Weber, J. Juliano, and A. Helser, "VRPN: a Device-Independent, Network-Transparent VR Peripheral System," in *Proceedings of the ACM symposium on Virtual reality software and technology*. ACM New York, NY, USA, 2001, pp. 55–61.

[14] P. Bellavista, A. Corradi, and C. Giannelli, "The Real Ad-hoc Multi-hop Peer-to-peer (RAMP) Middleware: An Easy-to-use Support for Spontaneous Networking," in *Computers and Communications (ISCC), 2010 IEEE Symposium on*. IEEE, 2010, pp. 463–470.