Performance Aspects of Data Transfer in a New Networked I/O Architecture

Cynthia Taylor
Computer Science and Engineering
University of California, San Diego
La Jolla, CA
cbtaylor@cs.ucsd.edu

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

***Abstract*—We present performance results of a new distributed I/O software architecture to support remote applications interacting with local I/O devices. The architecture emphasizes network transparency and ease of customization/extensibility in support of the vastly different needs of various applications and devices that can benefit from remote I/O. Networked I/O is achieved via a networked device driver that is split into two parts, one on each side of the network. 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. 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, one of the key problems of remote I/O, while remaining transparent to the application. We show that even with an implementation that operates almost entirely at user level (i.e., outside the operating system), good levels of performance that are adequate for even high intensity I/O, both in terms of efficiency and throughput, can be achieved.**

I. Introduction

In this work, we evaluate the performance of a new system software architecture where devices can communicate over the network with applications transparently, without applications having to be specifically designed for networked use. This is especially relevant to cloud computing, as it allows applications running in virtual machines in the cloud to easily communicate with I/O devices on a user's device. Legacy applications that were designed to run with devices locally are able to run in 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. Each side can be enhanced by transformation modules, which modify the I/O stream.

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 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.

II. Related Work

X-Windows and VNC are two classic thin client systems [1], [2]. Both of them allow remote I/O data to be forwarded over the network. However, both are designed to primarily support the standard devices of mouse, keyboard, and video data. In addition, in the case of X-windows, applications must be written specifically for the X-windows system in order to use it.

USB/IP sends device information at the driver level, allowing any USB device to work with the system [3]. This has the advantages of both transparency and full functionality. However, there is no way to change the behavior of different devices to account for the network or control how different updates are sent. THINC [4] and CameraCast [5] both use logical drivers in systems designed for video data being sent over a network. While both of these systems are focused on specific devices, the concept of a logical driver and the use of intermediate processing modules is important for remote I/O.

The classic work using modules to modify data between devices and applications is the UNIX Streams system [6]. More recently, there are many systems that create distributed environments for processing device input data for specific domains, where applications are written for these environments, and input data is passed through a series of processing "filters". These include the Berkeley Continuous Media

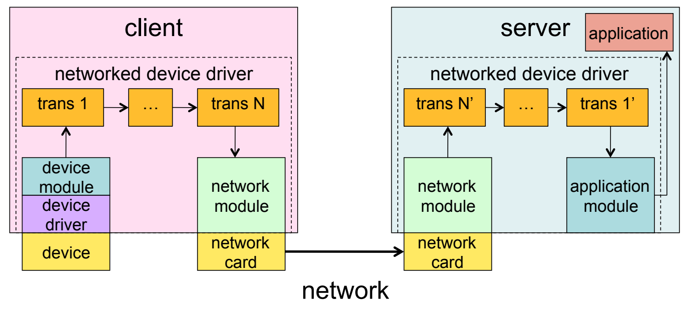

Figure 1. The networked device driver architecture. All device and network-related processing are encapsulated within the networked device driver, which operates on each side of the network.

Toolkit for distributed multimedia applications, Cascades for sensor networks, and a variety of systems for virtual reality applications, including Open Tracker [7], [8], [9], [10], [11].

We build on the ideas of these systems, integrating the network as a central object (whereby modules operate on each side of the network), and we use a mostly user-level implementation for maximum flexibility, portability and ease of implementation.

III. System Architecture

A. Architecture Summary

The central abstraction of our system is the *networked device driver*, shown in Fig. 1. By using the device driver as our central abstraction, we preserve transparency. Messages are created by either device or application, passed through half of the network device driver, which transforms them in some way, and sent to the network. Once across the network, they are read in by the network device driver, the transformations are reversed, and they are sent to their destination.

The system consists of four different types of modules. The *device communication module* collects messages from the device, packages them into the networked device driver message format, and forwards them to the next module. The *application communication module* communicates with the application, imitating the original driver for the device. *Network modules* transmit updates across the network. Optional *transformation modules* modify updates, changing them to compensate for the network. These modules are described in more detail in [12].

B. Transformation Modules

Messages travel through a set of optional transformation modules on either side of the networked device driver. These modules allow for the addition of extra functionality and compensation for the behavior of the network, while still preserving transparency. Each module performs a specific task, such as averaging, buffering, bundling, compression, encryption, etc. Modules vary in generality: some require specific syntactic or semantic knowledge about the messages they are transforming, and thus must be written for a specific device, while others are more generic and can work with any device.

Any change made to the format of a message on the client must be reversed on the server in order to preserve transparency. This undoing must be performed in the reverse order of the original transformations: if a message is encrypted and then compressed, it must be decompressed and then decrypted. Consequently, transformation modules are considered as *pairs*, consisting of an original transformation and its reversal. Pairs may exchange out-of-band messages, thus allowing for the communication of control information about how to process messages.

IV. Implementation

We wish to make it easy to create new networked device drivers, or customize them to suit a user's individual needs. To support this, our system is implemented almost entirely at the user level, to avoid running arbitrary code within the kernel, which is dangerous both because processes may create system failures on errors, and processes within the kernel may be able to maliciously affect the system. Each module is implemented as a separate process, allowing the use of the kernel's existing scheduler, and allowing processes to naturally block when there are no waiting messages for them to process. Our implementation is based on Linux, although the operating system mechanisms we rely on are found in most operating systems.

A. Implementation with Pipes

In our pipe-based implementation, all links between modules in the data stream are created with pipes. 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. 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. 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.

B. 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. 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 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 IV-A, but used purely for synchronization. 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.

C. *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 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). This becomes even more complicated when the message changes size over time. 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 V.

V. Performance

To demonstrate that our user-level implementation does not cause an undue amount of overhead, we tested two implementations that use different ways of passing data between modules, one using pipes and one using shared memory. All tests are 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, with a sample ping round-trip time of 0.235 msec.

A. *Performance of Pipes*

To test the overhead of adding additional modules using pipes, we created a transformation module that simply reads from an input pipe, and writes to an output pipe, performing no computation on the data (allowing us to focus purely on communication overhead). We used the video card network device driver to send various sized frames of video through a range of multiple copies of the simple transformation module. We instrumented the application communication module of the video card to measure how long it took to receive and display a frame. Since our transformation modules work in pairs, each additional transformation module is added to both sides.

In Figure 2 we show the average time it takes between transfers of a frame of video in both shared memory and pipe implementations, across varying numbers of transformation modules. Focusing first on pipes, adding transformation module pairs increases overhead, resulting in it taking longer to send and display a frame, as expected. The main performance dip is in adding the first transformation module: this adds 8.9% of overhead to the system with no transformation modules. Each additional transformation module adds an average of 2.3% 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.

B. *Performance of Shared Memory*

To test shared memory compared to pipes, we created a similar transformation module to that described in Section V-A, which simply passes the data through without modifying it. Using our shared memory implementation, this means the transformation module reads the starting point and size of the update in shared memory, and then writes it to the next transformation module, without touching or accessing any of the data in the shared memory. We again used the video network device driver to send video in a variety of frame sizes.

As shown in Figure 2, it is no surprise that the times in the shared memory implementation are essentially unaffected by adding transformation modules. Again, the biggest performance dip is the first transformation module, which adds 5.6% of overhead to the system. Each additional transformation module adds an average of 0.36%.

C. *Comparing Pipes and Shared Memory*

Focusing on the minimal times (with no transformation modules), for a 100 square pixel frame, it takes 2.92 msecs using pipes and 2.58 msecs using shared memory. For a 700 square pixel frame, it takes 166 msecs using pipes and 148 msecs using shared memory.

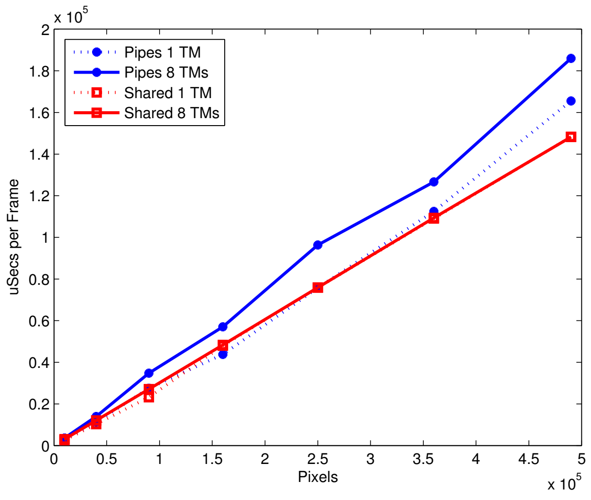

Figure 2. Performance of pipes versus that of shared memory. For each of the pipes and shared memory methods, measurements are shown for two situations: a single transformation module and eight transformation modules. Additional transformation module pairs increase the performance difference between pipes and shared memory by approximately 2%.

From the rest of the graph in Figure 2, one can see that additional overhead from using pipes grows with both the number of transformation modules and the frame size, though the slopes are not large. Without transformation modules, the system using pipes takes 6.7% longer to display a frame on average. Each additional transformation module adds 2.3% of overhead when compared to the shared memory implementation using the same number of transformation modules. This difference in overhead is a worst-case scenario since the transformation modules are simply transferring data. In more realistic 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.

VI. Conclusion

In this work, we presented a new networked device driver architecture to support remote I/O devices. To support network transparency, we encapsulate all networking and related processing inside the networked device driver. This means applications do not need to be modified to work with remote devices, making this a highly viable approach for cloud computing. Transformation modules, that form the key "programmable" aspect of the networked device driver, provide support for customizability while preserving transparency.

We evaluated two implementations of the networked device driver architecture, one where pipes are used to transfer data between all modules, and the other using 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.

References

[1] R. W. Scheifler and J. Gettys, "The x window system," *ACM Transactions on Graphics*, vol. 5, no. 2, pp. 79–109, 1986.

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

[3] 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 USENIX Annual Technical Conference*, 2005, p. 42.

[4] R. A. Baratto, L. Kim, and J. Nieh, "THINC: A virtual display architecture for thin-client computing," in *Proceedings of the 20th ACM Symposium on Operating Systems Principles (SOSP)*, 2005.

[5] 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.

[6] D. Ritchie, "A stream input-output system," *AT&T Bell Laboratories Technical Journal*, vol. 63, no. 8, pp. 1897–1910, 1984.

[7] K. Mayer-Patel and L. Rowe, "Design and performance of the berkeley continuous media toolkit," in *Proceedings of IS&T/SPIE Symposium on Electronic Imaging: Science & Technology (Multimedia Computing and Networking)*, 1997, pp. 194–206.

[8] 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*, 2006.

[9] G. Reitmayr and D. Schmalstieg, "OpenTracker: A flexible software design for three-dimensional interaction," *Virtual Reality*, vol. 9, no. 1, pp. 79–92, 2005.

[10] J. von Spiczak, E. Samset, S. DiMaio, G. Reitmayr, D. Schmalstieg, C. Burghart, and R. Kikinis, "Multi-modal event streams for virtual reality.," in *Proceedings of the 14th SPIE Annual Multimedia Computing and Networking Conference (MMCN'07), San Jose, California*, 2007.

[11] 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.

[12] C. Taylor and J. Pasquale, "A remote I/O solution for the Cloud," in *Proceedings of the 2012 IEEE International Conference on Cloud Computing (CLOUD)*, 2012.