A Remote I/O Solution for the Cloud

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*—With applications increasingly moving to the cloud, it is becoming common for an application to be separated by the network from the I/O devices with which the user is interacting. Currently this requires modifying the application to receive user input from the network rather than the device. We present a new I/O architecture in which the device driver is split into two parts, with the network between them. This architecture makes the network invisible to both device and application, allowing both of them to work unmodified. Our architecture also supports transformation modules, each of which comes in a pair that operates on each side of the network. Via these module pairs, the resulting system is capable of supporting the modification of the I/O stream in a variety of ways to compensate for the network, while remaining transparent to the application.**

I. Introduction

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. By this we mean obtaining inputs or delivering outputs to I/O devices that are separated from the application by a network. The I/O devices are typically located where the user is located, whereas the application is located where the computing resources are, somewhere in the cloud.

Nowhere is this issue more evident than in moving legacy applications to the cloud, including applications that interact with a wide variety of I/O devices. The traditional approach to solving this has been thin clients, but thin clients typically support only a specific subset of devices, generally limited to a mouse, keyboard, and video. To support easily moving existing applications to the cloud, we require a flexible solution that will allow us to transparently interpose the network between any application and device.

Currently, moving applications that work with exotic devices across the network may require completely rewriting the application to take input from the network, rather than directly from the device driver. If the source code is not available, this involves tactics such as automatically rewriting the GUI and manually translating and forwarding user input[1], [2]. Supporting network transparency means existing applications can interact with remote devices without changes. This avoids having to rewrite applications for the cloud and related issues such as parallel code maintenance. This also allows us to easily move legacy applications, for which source code is often not available, into the cloud.

Moving applications to the cloud introduces many factors which can degrade a user's experience with an application. The presence of an intervening network can introduce delays, bandwidth restrictions, and uneven transmission rates that result in problems such as jitter. The network may also be insecure. Consequently, we require a solution that allows for flexibility and mediation, one where the data being transmitted can be modified between the application and device, performing operations such as compression and encryption, and yet, maintaining transparency as a primary goal.

In this paper, we present a system architecture that supports transparency to the device and application, while still allowing the transformation of the data passing between them. We support transparency with 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.

To transform the data stream, our architecture supports transformation modules that enact some change on the data as it is passed through them. To preserve transparency, these modules are created in pairs, one on either side of the network. A simple example of coordinated pair operation is where one module compresses the data and its corresponding module decompresses the data. More sophisticated examples involve cooperative send-side and receive-side buffering, bundling and unbundling of data blocks, and bandwidth-reducing averaging in response to dynamic networking conditions. Thus, we are able to both process data to compensate for the network and forward data to the application in its original format.

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 [3]. Since our architecture virtualizes the device driver by splitting it into two parts, the application half of the driver can run

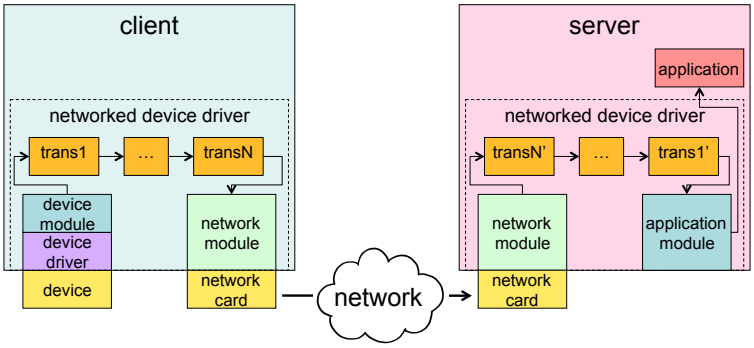

Figure 1. The networked device driver system architecture showing how I/O data flows through networked device driver. The data stream is sourced at the device and sinked at the application. A set of transformation modules transform the data stream, and come in pairs (e.g., trans*x* and trans*x*').

entirely within the virtual machine, without the hypervisor having to virtualize the device.

II. System Architecture

A. Architecture Summary

At its most basic level, the system architecture must support the passing of messages between a device and an application, each on a different machine, communicating over the network. To preserve transparency, we encapsulate all network and network-related operations within a *networked device driver*, as shown in Fig.1. Messages are created at either the device or application, passed through one half of the networked device driver, possibly transformed in some manner, and sent over the network. On the other machine, they are read from the network, the transformation, if any, is 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 translating between the application and the networked device driver. Similarly, the *device communication module* translates between the device and the networked device driver. *Network modules* send and receive messages over the network. Lastly, optional *transformation modules* (to be discussed in Section III) modify messages as they are sent through the system, making changes to them, e.g., to compensate for the effects of the network.

Throughout this paper, we refer to the machine with the I/O devices as the client, and the application machine as the server. We also refer to the originator of the updates (either device or application) as the "source," and the recipient of the messages (application if the source is the device, device if the source is the application) as the "sink."

B. Data Streams

We refer to the main flow of communication in a networked device driver as a *data stream*. The data stream consists of updates that travel between the device and application, as illustrated in Fig.1. 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 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 reverses the modification applied by its matching transformation modules on the source machine. The updates are then sent to the communication module, which feeds them to the sink.

Our system allows messages to be modified/repackaged 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 an asymmetric fashion allows the system to handle each data stream in a way that best fits its unique data profile.

C. Header Format

Each message in the I/O stream is encapsulated in a *container* [4], 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. Headers for specific transformation modules are 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.

D. 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. The raw driver may be strictly a kernel-level driver or it may include a user-level component. The raw driver operates normally, communicating by either providing information to a register for polling, or by generating interrupts.

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.

E. Application Communication Module

The last transformation module passes the update to the *application communication 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 consists of both a user-level component and a kernel-level component. The kernel-level component is necessary for the application to 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.

F. 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, can be used with any device, and are parameterized so that an appropriate standard network protocol can be used. 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 is a natural fit. Modules for new or experimental network protocols may also be created.

III. 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.

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.

1) Transformation Module Pairs: 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. Notice the resulting ordering this implies in the transformation modules in Fig.1. Some transformation modules (e.g., averaging) change the content of the message rather than its form, and thus may have no reverse of their action. Each such module is paired with a "no-op" module.

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 are 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 falls 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. For example, in our prototype, 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. In our prototype, 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 create transformation modules which multiplex data in appropriate ways. For example, depending on video input is being processed, we may want 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 are 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.

Synchronizing Multiple Data Streams: For applications that receive data from multiple devices, it may be necessary to synchronize the updates from these devices. This is 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.

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.

3) Out-of-Band Messages: All transformation modules support out-of-band messaging, i.e., the use of a separate communication "control" channel in addition to 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 sends 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.

IV. Implementation

One of our design goals is to make it easy to customize networked device drivers or create new ones. We implement at the user-level whenever possible, as this avoids the dangers of running arbitrary code in the kernel, both in terms of processes creating system failures on errors, and processes being able to maliciously effect the system. Running at the user-level also allows us to leverage existing mechanisms. We use separate processes for each module, which allows us to use the kernel's built in scheduling mechanisms, rather than having to create our own scheduler, and allows processes to naturally block when there is no more data for them to process. All device communication modules, network modules, and transformation modules run at the user-level.

We only depart from the user-level when necessary to preserve transparency. Within the application communication module, we sometimes use kernel modules to create the device driver interface that the application is familiar with, as illustrated in Fig.2. Adding this kernel module at the very end allows us to preserve transparency, while still

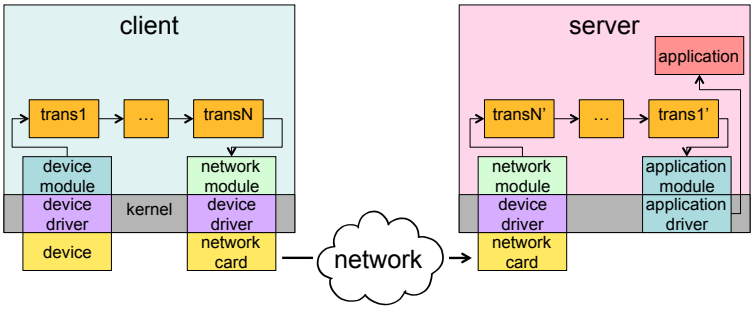

Figure 2. Only the application driver and raw driver run within the kernel.

giving us all the advantages of running at the user-level for the rest of the system.

We aim for this system to be as extensible as possible, so device makers, application creators, and users can all extend it 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.

A. Space Navigator

For our initial implementation, we created a networked device driver for a 3D mouse device called the Space Navigator [5]. 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.

To create a device communication module for the Space Navigator, we create a user-level process that receives events from the Space Navigator, processes these events into our update format, and then writes the update(s) to a pipe.

The Space Navigator uses a user-level device driver, so to create an application communication module for it, we rewrote the device driver to receive updates from a pipe rather than from the device. This was a simple change, and required very few changes to the structure of the driver. Because this is a modification of the existing driver, and the driver continues to raise events in the exact same way, no modification is needed for any application to use our version of the driver.

B. Mouse and Keyboard

We created networked device drivers for the basic mouse and keyboard. In linux, the mouse and keyboard both use evdev events, so we created the modules for them that work in very similar ways [6]. For the device communication modules, we create a user level process that reads from the driver for the device, and captures the events. We then create an update with the information from the event, and write it to a 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 the pipe, and writes it to a sysfs node which maps to the kernel module. The kernel module then generates the evdev event for the update. Because we used a 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.

C. Video Card

We access the video card through the frame buffer, the area of memory that holds the pixel values for the display. To create the device communication module, we developed a user level process that reads the framebuffer values, and then writes them to a pipe. Likewise, for the application communication module, we read in pixel values from a pipe, and then write them to the frame buffer. The resulting framebuffer values are the used by the display.

D. Network Modules

We have implemented network modules that can use either TCP or UDP (but can easily be extended to use other network protocols). The network module on the device side simply reads an update from a pipe and sends it over a socket to the IP and port specified. Likewise, the network module on the application side reads an update from a socket, and writes it to a pipe.

V. 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 that in mind, we test the effects of pipes specifically, as well as a test of the system as a whole.

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.

A. Update Speed of Networked Device Drivers vs Standard Drivers

To calculate how much delay our system added to the time it takes an application to receive an update, we instrumented our system and measured. Since the update takes a one way path over the network, we measured the system in three discrete parts to get an accurate measurement. All measurements were taking using the rtdsc and rtdscl registers to determine the number of clock cycles that had passed, and converting from clock cycles into milliseconds. We averaged over 2000 tests for all measurements.

On the driver side, we measured the time it took between when the device communication module received an update, and when the network module was ready to send it over the network: this took an average of 36 microseconds, with a standard deviation of 9 microseconds. For the network, we needed to measure a round trip time 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 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. We then divided the resulting measurement by two to get the one-way time, resulting in an average time of 380 microseconds with standard deviation of 14 microseconds. (A ping taken at the same time had a one-way average time of 120 microseconds.) On the application side, we measured 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, for an average time of 17 microseconds with standard deviation of 15 microseconds. The total time that our system added to the update was 433 microseconds. The Space Navigator operates as a rate of 62.5 Hz, or every 16 ms, so 433 microseconds will not add a perceptible delay.

B. The Networked Device Driver Compared to VNC

To show how our system compared to custom device-forwarding applications, we compared a networked device driver built for the video card to the classic thin client VNC. We measured their performance displaying a video of a television show, running at 24 frames per second. We instrumented both VNC and the networked device driver 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. We instrumented the client 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. We added code to this module to measure how long it takes for each cycle of reading and updating. We averaged over 2000 updates.

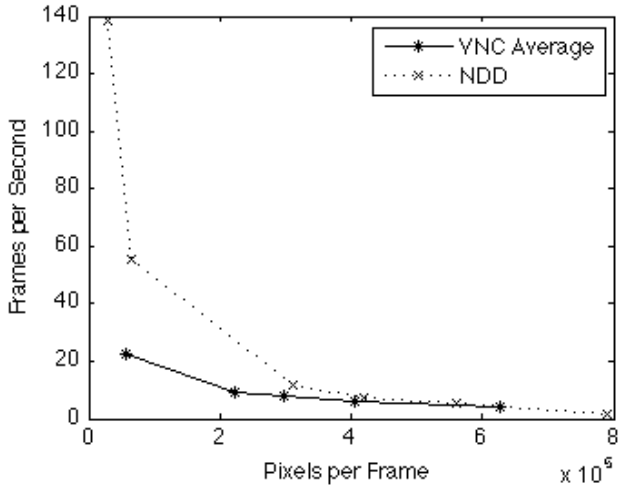

Figure 3. The video card networked device driver compared to VNC.

In Fig.3 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, 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. As we increase the average numbers of pixels per frame, 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.

C. Adding Transformation Modules

A key component of our system is the ability to add transformation modules to process updates for the network. Our system is implemented using pipes to transfer data between modules. In this experiment, we demonstrated that adding transformation modules does not create significant overhead due to memory copies. We compared the networked device driver running the compression module to the

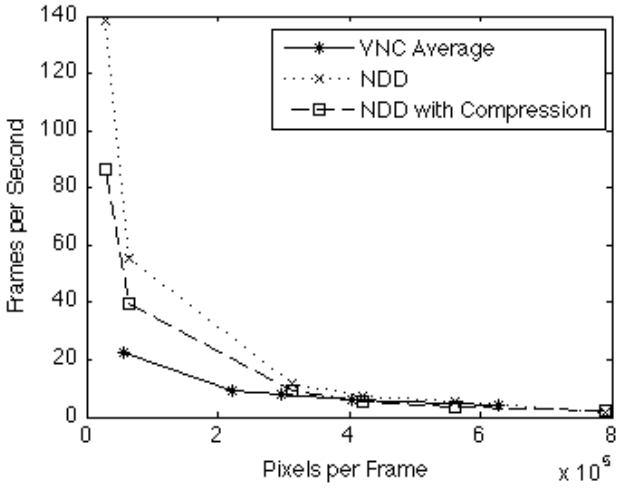

Figure 4. The Video Card Networked Device Driver, with and without compression, compared to VNC.

networked device driver without it. 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 wish 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 two frames per second. The frame rate of the networked device driver with compression also 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.

VI. Related Work

The two classic approaches to sending I/O over the network are demonstrated by X-Windows [7] and VNC [8]. In X-Windows, the high level commands sent to the video driver by the application are sent over the network, and interpreted by the video driver on the local machine. A drawback is that applications must be specifically written for X-Windows in order to work with it. In VNC, the pixel data is copied from the framebuffer and sent to the client, where it is rendered by a user-space application. While this 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 (the same is true for X-Windows).

Work on USB over IP has focused on sending device information below the driver level [9]. 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.

THINC [10] and CameraCast [11] both use logical drivers in systems designed for video data being sent over a network. CameraCast also allows users to create intermediate modules to process the video being sent. However, both of these systems are focused on specific devices (mouse, keyboard, video card, and audio card in THINC, webcams in CameraCast), and the THINC system does not allow for users to add application-specific modules to process data.

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 Toolkit for distributed multimedia applications, Cascades for sensor networks, and a variety of systems for virtual reality applications, including Open Tracker [12], [13], [14], [15], [16].

Other systems have been designed with the idea of being able to modify I/O data. The classic work is the UNIX Streams system [17]. Plan 9 builds on UNIX Streams, but with better integration for the network and its 8 1/2 windowing system [18], [19].

Ishii et al. describe a datastream approach to processing sensor data in the cloud [20]. They use this approach to distribute pieces of the processing application throughout the cloud. This is significantly different from our system, where we use a data stream approach to process device data before it reaches the application, and use module pairs to ensure the data reaches the application in its original format.

Finally, Yang et al. describe methods for improving both performance and fault tolerance of messaging in cloud system [21], [22]. Such methods might be useful in the remote I/O system we describe.

VII. Conclusion

In this work, we presented the networked device driver architecture to support remote I/O devices. This architecture is especially relevant to the cloud, as it allows applications running in virtual machines in the cloud to easily communicate with I/O devices on a user's device. To support network transparency, we encapsulated all networking and related processing inside the device driver. This means applications do not need to be modified to work with remote devices.

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. To support customizability while preserving transparency, 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.

In our experiments, we showed that converting a device driver to a networked device driver, with a significant portion of the implementation operating at user level, does not cause excessive overhead. We also showed that a networked device driver created for the video card is competitive in performance with VNC, a thin client device developed and optimized for forwarding video over the network. Lastly, we showed that adding transformation modules to our system greatly extends flexibility. We demonstrated that our system allows for both network transparency and customized processing of the data stream to compensate for the network, without a significant performance burden.

REFERENCES

[1] 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*. IEEE, 2011, pp. 750–751.

[2] C. Phanouriou and M. Abrams, "Transforming command-line driven systems to web applications," *Computer Networks and ISDN Systems*, vol. 29, no. 8-13, pp. 1497–1505, 1997.

[3] C. Waldspurger and M. Rosenblum, "I/o virtualization," *Communications of the ACM*, vol. 55, no. 1, pp. 66–73, 2012.

[4] J. Pasquale, E. Anderson, and P. Muller, "Container shipping: operating system support for i/o-intensive applications," *Computer*, vol. 27, no. 3, pp. 84–93, 1994.

[5] "3d Connexion Space Navigator." [Online]. Available: http://www.3dconnexion.com/

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

[7] R. W. Scheifler and J. Gettys, "The x window system," *ACM Trans. Graph.*, vol. 5, no. 2, pp. 79–109, 1986.

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

[9] 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.

[10] 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.

[11] 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.

[12] 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.

[13] 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.

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

[15] 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.

[16] 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.

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

[18] D. Presotto, R. Pike, K. Thompson, and H. Trickey, "Plan 9: A distributed system," in *Proceedings of Spring 1991 EurOpen*, 1991, pp. 49–56. [Online]. Available: citeseer.ist.psu.edu/presotto91plan.html

[19] 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*. Berkeley, CA, USA: USENIX, 1991, pp. 257–265 (of x 473). [Online]. Available: citeseer.ist.psu.edu/article/pike91plan.html

[20] A. Ishii and T. Suzumura, "Elastic stream computing with clouds," in *Cloud Computing (CLOUD), 2011 IEEE International Conference on*. IEEE, 2011, pp. 195–202.

[21] C. Hsu, A. Cuzzocrea, and S. Chen, "Cad: an efficient data management and migration scheme across clouds for data-intensive scientific applications," *Data Management in Grid and Peer-to-Peer Systems*, pp. 120–134, 2011.

[22] C. Yang, W. Chou, C. Hsu, and A. Cuzzocrea, "On improvement of cloud virtual machine availability with virtualization fault tolerance mechanism," in *Cloud Computing Technology and Science (CloudCom), 2011 IEEE Third International Conference on*. IEEE, 2011, pp. 122–129.