The Proximal Workspace Architecture for Wearable I/O-Device Cloud Applications

Cynthia Taylor
Computer Science
Oberlin College
Oberlin, OH
ctaylor@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 enhanced cloud computing architecture, called the Proximal Workspace, to allow access and interaction between lightweight wearable I/O devices, e.g., video glasses, earphones, wrist displays, body sensors, etc., and ubiquitous applications that represent a new generation of computation-and-data-intensive programs. While wearable devices offer an easy way for these applications to collect user data and offer feedback, the applications cannot be run natively and completely on these devices because of high resource demands. Making these applications available via a cloud, while promoting ubiquitous access and providing the necessary resources to execute the applications, induces large delays due to network latency.

The Proximal Workspace provides nearby computing power to the user's devices and thus mediates between them and the cloud's computing resources. The workspace is designed to run any subset of activities that cannot be run on a user's device due to computation speed or storage size, and cannot be run on a cloud server due to network latency. We also describe a powerful abstraction for networked communication, called Networked Device Drivers, which provide the underlying communication support for Proximal Workspaces in a way that promotes simplicity and transparency.

1. INTRODUCTION

Promoters of ubiquitous computing have long dreamed of a time where computer applications will fade into the background, and users will no longer sit at desktops, or painstakingly navigate clunky screens on bulky portable devices. Instead, collections of sensors will unobtrusively collect information on the user and their surroundings, and applications will process these inputs and modify the users' surroundings based on their learned preferences [19]. Today we have access to lightweight wearable devices, an unprecedented number of sensors, and machine learning applications which are capable of developing and adapting complex models of behavior.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
CSSWearable 2014, December 02-04, London, Great Britain
Copyright © 2014 ICST 978-1-63190-039-6
DOI 10.4108/icst.mobiquitous.2014.258037

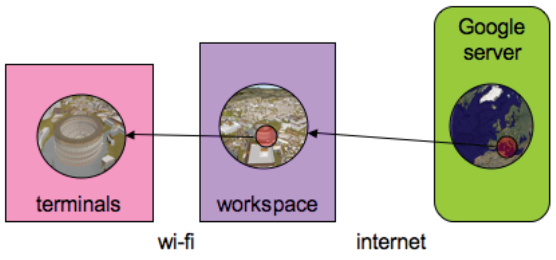

Figure 1: The workspace stores data that is too large to hold on the lightweight wearable I/O device.

To begin to realize this vision of ubiquity, we need applications to take full advantage of the new data sources these sensors offer, requiring intensive data processing and machine learning that cannot run natively on low-powered wearable devices. The traditional response to this has been to move the required computation from the local device into a server farm somewhere (i.e., "the cloud"). However, this approach can suffer from network latency, which can introduce unwanted (and intrusive to users) delays in response time. In this work, we address this problem by introducing the concept of a *Proximal Workspace* (or simply "workspace" for the remainder of the paper), a low-latency intermediary between the client and server.

In this paper, we consider the following computing environment: A user has some collection of input/output devices. An application receives input from these devices, and uses this to construct a model of what the user is doing and what their surroundings are like. It then sends appropriate feedback to the user's devices, and other devices in the user's immediate surroundings. This model is inspired by the idea of *wearable space*, which is described by Samdanis et al as "a ubiquitous technologies environment which through wearable and spatially embedded interfaces links the human body to architectural spaces" [9]. This wearable space incorporates I/O devices both worn by the user and embedded in their surroundings to create a ubiquitous environment, in which information about the user is constantly being gathered and combined with previous information about their preferences to alter their immediate surroundings.

We see the wearable space concept as presenting a new model for how data is sent between a client and server, and investigate the challenges presented by this new model. In the traditional client-server model, the client will usually request a specific piece of data from the server (e.g. a webpage, image, or multimedia file), and the server will send the client that specific item. In this new model, the client will send the server a collection of device updates that represented the user's current state, and then the server will send back a large amount of information which will be interpreted and displayed by the output devices both on and about the user. The user can then interactively explore this data once it is stored on the client, and the client will periodically send new information about the user to the server.

Previously, the idea of sending surrounding data has been used as an optimization for slow data transfers, i.e., pre-fetching surrounding rows in a database. In the wearable space model, we see two issues that make things different and that present new challenges. The first is the nature of the data: a typical ubiquitous application may send vast amounts of very detailed multimedia data to the client. The second is the nature of the user's exploration: typically, a user will explore the information in a non-linear fashion. These two factors mean that the client has to deal with the data it is receiving in a very different way than it did in the past. The amount of data transferred requires significant storage capacity on the device, and the exploration by the user will create a significant amount of on-demand processing.

Moving applications to a server within the cloud in a thin-client fashion induces large delays due to network latency. To solve this problem, we have designed a new system architecture whose key feature is the addition of a workspace as a low-latency (relative to the client) intermediary between a client and server(s). Figure 1 illustrates how the workspace fits in to this new data model.

2. RELATED WORK

In this work we present a system to support applications that combine both wearable and embedded I/O. Related to this, Samdanis *et al* introduce the idea of *wearable space*, a combination of wearable and spatially embedded I/O devices that create a ubiquitous computing environment [9, 8]. Wearable space builds on both work in both wearable computing, and in interactive architecture. The architecturally embedded devices can be as simple as speakers and video displays, or as complex as the digitally moveable walls proposed in Synthetic Space [12].

To support these systems, we propose the workspace architecture, in which low-powered wearable and embedded devices use a proximal servers to offload computation. There are a number of systems that incorporate the idea of using physically local resources for computation, although none of them focus on interaction with I/O devices.

The Slingshot system [11] assumes a model where there are computational resources available at WiFi hotspots. The user runs a first-class replica of their computing session on their home server, and second-class replicas are instantiated on available servers near the client device. A client proxy is responsible for locating local servers, deploying second-class replicas, and coordinating their replies. Each replica runs in its own virtual machine.

In their work on Cloudlets, Satyanarayanan *et al* also argue for a proximity-focused architecture. Their system architecture approach is to use virtual machines to support remote processing[10]. This builds on Internet Suspend/Resume system [6], which combines virtual machines and distributed file systems with the goal of "mobile computing that mimics the opening and closing of a laptop, but avoids physical transport of hardware." It uses a Virtual Machine Manager to suspend the virtual machine in which processes are running, and save the state to disk. It then uses a distributed file system as a transport mechanism to move the files containing the state from one machine to another.

In Mobidesk [4], network and operating system virtualization is added to a thin client to create a system where user sessions can be transparently moved between servers. Mobidesk uses THINC [3] for its display virtualization, and builds on Zap [7] for its operating system virtualization, with each session being created inside a pod. It is designed as a proxy-based server cluster system, with the client located anywhere outside the proxy, and the proxy providing all access to a network of servers on which sessions run.

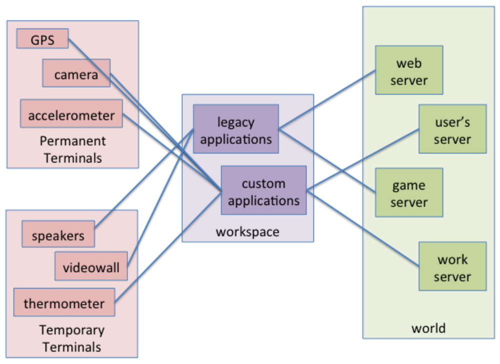

Figure 2: The system introduces a workspace between the client and servers.

3. ARCHITECTURE

The Proximal Workspace Architecture, the original version of which we described in [14], is comprised of three parts, illustrated in Figure 2: (1) *terminals*, the (wearable) sensors and devices that the user actually interacts with and that provide data about the users location; (2) the *world*, consisting of the user's home and/or work computers, web servers, game servers, and anything else the user interacts with through the internet; and (3) the *workspace*, a temporary computing session running on computational resources very close to the terminals, capable of extending functionality of both terminals and the world. We now describe these classes of components in more detail.

Terminals consist of I/O devices worn or carried by the user, as well as sensors in the room surrounding the user. They are used for input and output. Input can be specific user actions, equivalent to mouse movements and key presses, or it can be information from sensors such as video cameras, accelerometers, GPS units, thermometers, etc. The terminals may include a hub for local communication and coordination. A set of terminals could consist of a smartphone with an integrated GPS and camera, with a wall display in the room the user is in. Terminals form a component similar to the client in traditional thin client systems. Terminals are not required to be capable of anything more than capturing input, displaying output and communicating with the rest of the system. Terminals attached to the user can be thought of as *permanent terminals* that will persist throughout a session. Terminals attached to the room are considered *temporary terminals* which may not persist through an entire session.

The "world" is the set of servers that communicate with the rest of the system through the internet. These can include a reference website the user is visiting for the first time, or a system server with which they are in semi-constant contact. All persistent state is stored within the world. We make no assumptions about the latency between the terminals and any given server within the world. Some parts of the world may be provided by others and contain large numbers of factors we cannot control. How the world interacts with the rest of the system can have a large impact on system performance and user experience. How long a part of the world remains in the system can very from seconds to decades, and parts can remain consist over different workspace sessions.

We define the workspace more precisely as a set of resources, both hardware and software, provided to the user by the workspace server, a machine very close to the user in the network designed to provide systems with these resources. The workspace is used to aid in applications that depend on the users locality, and applications that must quickly communicate with the terminals. In terms of hardware resources, the workspace provides local power for computing, memory, and storage, creating the illusion that the terminals have much more power than they do in actuality. It is aware of its physical location and may be optimized for tasks that are frequently performed at its location. A set of terminals interacts with only one workspace at a time, but may include several different workspace sessions over a given time period.

As for software resources, the workspace provides a set of middleware utilties in the form of helper applications, generally designed to do tasks that benefit from low latency but are too computationaly intensive to run on the terminals. Examples of these sorts of tasks include rendering, caching data, and prefetching data. These applications may be legacy applications, such as VNC, or new applications created just for these tasks. These helper applications may be dynamically uploaded (not arbitrary code, but code that has been verified, from a trusted repositary) to the workspace, where they are then executed and cached for future use.

The workspace can cache information that relates to its physical location, e.g., map data, location dependent game elements, etc, and either forward it to the client or access it with internal utilities, creating performance improvements based on its constant geographic location. Additionally, as the user travels and changes workspace servers, the set of *temporary terminals* will also change. The workspace understands its set of temporary terminals and their abilities, and has the ability to change which set of terminals it is sending information to, so it can send video output to a wall-mounted monitor when the user is in proximity, or to the user's phone display while the user is mobile.

During times when the user is away from a location with a workspace infrastructure, the workspace may run on a device like a phone, resulting in lower performance, but still enabling the user to send information about themselves to the world, and receive updates from the world. During these times, more computation may be offloaded to the world. When the user returns to a location with available workspace servers, the workspace can migrate back to a more powerful server, enable faster and richer processing.

The ability for the workspace to remotely communicate with the terminals is provided by the Networked Device Driver architecture, described below. Specific modifications of the NDD architecture to allow it to support the workspace system are described in section 4.9.

4. NETWORKED DEVICE DRIVERS

A key middleware/operating system level mechanism to support the proximal workspace architecture is the *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 workspace server. 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.

4.1 Architecture

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. To preserve transparency, we encapsulate all network and network-related operations within a *networked device driver*, as show in Fig. 3. 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.

4.2 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 Fig. 3. 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 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.

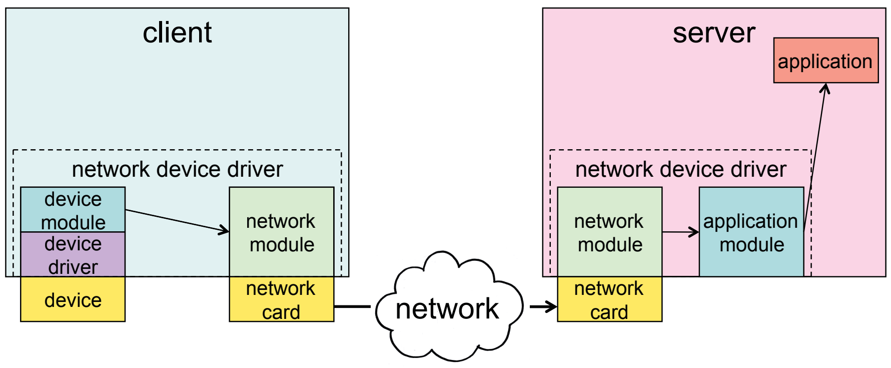

Figure 3: This data stream is sourced at the device and sinked at the application.

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.

4.3 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.

4.4 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 may be a natural fit. Modules for new or experimental network protocols may also be created.

4.5 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 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.

4.6 Transformation Modules

The data stream passes through a series of optional transformation modules before it reaches its 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.

4.6.1 Transformation Module Pairs

To preserve transparency, any change made to the format of the message must later be "undone." If the data stream 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. To preserve such ordering, transformation modules act in pairs, with one half of the pair performing an action on the client, and the other half

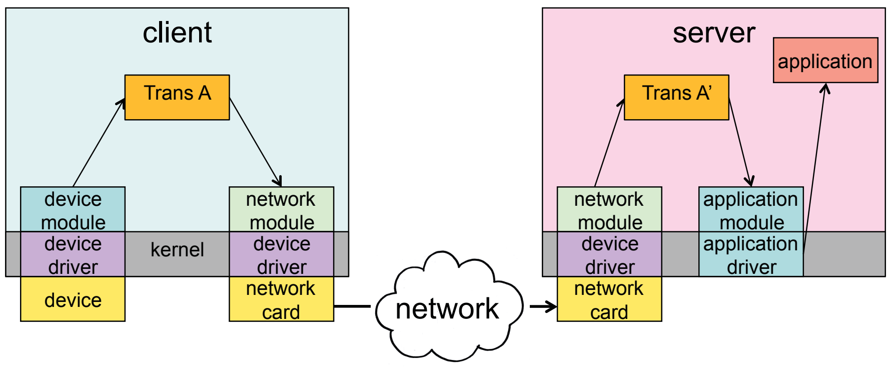

Figure 4: Only the application driver and raw driver run within the kernel.

reversing it in the server. These pairs are arranged in a nested fashion, so their order on the server is the reverse of their order on the client. 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.

4.6.2 Functionality

The type of additional functionality required will vary for each application. A list of possible functions that illustrate the potential uses for the system include:

- Averaging
- Buffering
- Bundling
- Compressing
- Discarding Non-Recent Updates
- Encrypting
- Pre-Fetching
- Synchronizing Multiple Data Streams

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. Finally, in addition to compensating for the network, some applications may wish to extend their functionality by performing modifications to the I/O stream.

4.6.3 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.

4.7 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 existing 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 on 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 Figure 4. 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.

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, a designer must provide 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.

We have implemented Network Device drivers for the mouse, keyboard, and video card, as well as for the Space Navigator, a 3D mouse device which produces updates with 3 translation values and 3 rotation values [1]. We have also implemented network modules for TCP/IP.

4.8 Basic Performance of NDD

As a basic measure of performance focusing solely on Networked Device Drivers (experimental results in an environment using the Proximal Workspace are presented in Section 5), we determined how long it takes a message (of various sizes) to travel along the data stream from the source to the

Table 1: 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

sink, using the most basic of network device drivers. This tells us how much latency our system is adding to the device, and helps determine how much overhead the system incurs. 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. Both machines were running Ubuntu Linux, and had wired connections to a relatively fast campus network, with a sample ping round-trip time of 0.235 msec.

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 created a synthetic device module which produces a random character array in a specified size every two seconds. Using this setup, we measured the end-to-end time for a variety of message sizes. 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. 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.

As shown in Table 1, the results show that the 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.

For more on the Networked Device Driver architecture, including more extensive performance results, see [17, 15, 16].

4.9 Applying NDD to Wearables

In an effort to apply and extend the existing NDD architecture to provide more support for both wearables and wearable spaces in general, and the workspace system, we present the following extensions.

4.9.1 Supporting Changing Terminals

As a user changes locations, their collection of *temporary terminals* may change. To support this, the workspace can check what terminals are available to it at the start of a workspace session, and reconnect to the appropriate set of terminals, using *permanent terminals* if the appropriate temporary terminal is no longer available. To support this, each workspace server will have a standard classification for each of its temporary terminals, such as "video output", "sound input", etc.

4.9.2 Translating Updates For New Devices

Wearable devices afford us the opportunity to work with a radically different set of input and output devices than are traditionally used in the desktop. When working with these new devices and legacy applications, the NDD system must translate messages from these new devices into the more traditional forms that desktop applications were designed to use. In some cases, this may be relatively simple, such as dealing with a smaller screen by either scaling down the original image, or only showing a portion of the screen. However, there may also be more complex transformations, for example translating an audio signal to a haptic signal, in a manner similar to setting one's cellphone to vibrate. With additional transformation modules designed for this sort of translation, this kind of adapting and translating signals from new devices to applications not designed to work with them would be easy to accomplish within the NDD framework.

4.9.3 Embedded Devices

Consider a simple device consisting of some collection of devices which communicate with the user (screens, microphones, LEDs, GPS, vibration unit, accelerometer, etc), a microcontroller, and some device which communicates with a local network (wifi, bluetooth, etc). Such a device could be extremely small and lightweight, but would be able to do very little actual computation on the device. By running a Networked Device Driver on such a device, we could move all computation from the device itself to the local workspace and remote servers, as well as giving it the ability to work with legacy applications.

With this in mind, we plan to port a simplified version of the NDD architecture to the Arduino microcontroller platform [2]. Since running on a microcontroller will not allow us to easily use multiple processes, this will likely involve developing a compiler to create a single process version of a network device driver using arbitrary transformation modules.

5. EXPERIMENTAL RESULTS: GOOGLE EARTH/ANCIENT ROME 3D

An important goal is to have utilities that work in conjunction with unmodified applications, rather than rewriting applications to work within our system architecture. Building things to work with unmodified applications offers many advantages to developers and users, including ease of installation and avoiding parallel code maintenance. We have taken various approaches to making our system work with unmodified applications, including virtualization-based approaches such as remote display applications and running applications in virtual machines, and intercepting messages sent between client and server applications. An example of the second approach can be seen in our previous work, Improving VNC Performance, in which we add a Message Accelerator which sits between the VNC client and server [13]. No modifications were required to either than client or server to add the Message Accelerator the client behaves as though the Accelerator were the server, and the server behaves as if it were the client.

To demonstrate the usefulness of the workspace architecture, supported by networked device drivers, we have adapted Google Earth Ancient Rome 3D with the goal of the

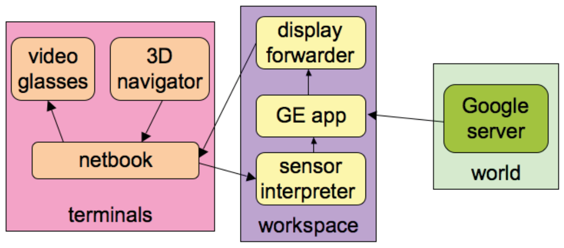

Figure 5: Google Earth Ancient Rome 3D adapted for the Proximal Workspace Architecture

user being able to intuitively navigate through renderings of Ancient Rome in video glasses, without being hampered by any bulky equipment. Google Earth is a perfect example of an application which fits our new data model. The user interactively explores a rendering of ancient Rome, either with a keyboard and mouse or with a more sophisticated input device, and while they explore the Google Server periodically sends a very large amount of multimedia data describing the area they are exploring to the client, which then renders it on a frame-by-frame basis. This puts two burdens on the client: it must be able to store the vast amounts of data being sent, and it must be able to quickly render a complex scene. Google's suggested minimum specs for computers to be able to display Ancient Rome 3d include 512 MB RAM, 2 GB of free disk space, Network speed of 768 Kbits/sec or better, and a 3D-capable video card with 32 MB of VRAM or greater. This is beyond the capabilities of a lightweight wearable I/O "glasses" device, but by adapting Google Earth Ancient Rome 3D to our system model, we can allow a user to use it on such a device.

In adapting Ancient Rome 3D to the workspaces architecture, we considered both where parts of the application should be distributed, and what utilities must be created to aid the distribution. The hardware for our test system consists of a Dell Optiplex 755 with a Radeon X1300 video card with 256MB of RAM running Windows XP as the workspace server, with a Lenovo Ideapad S10e Netbook running Ubuntu connected to a 3d Connexion Space Navigator and Video Glasses as the terminals. The Space Navigator is a joystick-like device that records both rotation and pressure around the x, y and z axes [1]. Since the Google Earth application cannot meet real time performance demands while running on the netbook, we moved it to the workspace. Once Google Earth is running on the workspace, we must create utilities to forward input from the netbook to Google Earth, and forward the display updates from the workspace server to a netbook. This is illustrated in Figure 5. To forward the display, we use TightVNC [18]. To forward input from the Space Navigator, we run a program on the netbook which forwards the raw input from the Space Navigator to the workspace server, where it is then aggregated, translated into units appropriate for Google Earth, and sent to the Google Earth application via API calls.

For our initial results, we measured the frame rate of Google Earth Ancient Rome 3D running natively on the Lenovo ideapad using the Fraps video benchmarking tool [5]. Running natively, Google Earth displayed 0.16 frames per second, resulting in a virtually unusable application. We then ran the adapted Google Earth Ancient Rome 3D on our system, with a link between the server and netbook that had an average round trip time of 0.42 ms, measuring our frame rate by using an instrumented version of Thin VNC. Our version achieved an average frame rate of 7.08 frames per second, resulting in an easily usable application and pleasant user experience.

By adapting Google Earth Ancient Rome to the Proximal Workspace architecture, we allow users to use the application on a lightweight netbook, something that it would be impossible to do running the application natively. Because of the low latency connection between the workspace server and the terminals, the performance of the application is quite good, with no lag between the display of frames. To the user, the effect is as though they were running the application on a much more powerful system, but without being tethered to a desktop machine.

6. CONCLUSIONS

We have presented the Proximal Workspace Architecture to support wearable spaces. Along with the Networked Device Driver, this architecture supports ubiquitous applications that communicate through light-weight wearable and embedded devices, and are highly computationally intensive. As these applications are highly interactive, it is imperative that they still execute "near" the user. Consequently, a dynamically allocated workspace that provides computational and memory resources, that is proximal to the user, and that supports communication with arbitrary I/O devices is the novelty of our design.

In addition to building individual utilities for this architecture, our current work is to also explore how to best design the system as a whole. Moving computation from the client to the workspace adds new issues that must be solved. For example, the workspace must be able to correctly save persistent data at the end of a session with the client, which means it must have a mechanism for figuring out which data should be saved, and which server in the world to save it to. There must be a mechanism for the client to discover and be assigned to a workspace server which is capable of handling its applications. Multiple client sessions within the same workspace machine raises issues of security, privacy, and QoS scheduling.

Based on our existing work with Google Earth, the Proximal Workspace system architecture offers unique performance advantages for ubiquitous applications and systems, especially those that use the cloud. By adding a workspace, systems can allow users to carry only lightweight wearable equipment, but avoid the costly performance lag of using a pure thin client system with the cloud. In addition, the location-based nature of the workspace system is an ideal match for wearable spaces applications, allowing workspaces to cache information about their own locations and serve it quickly to visiting or embedded terminals.

7. REFERENCES

[1] 3d Connexion Space Navigator. http://www.3dconnexion.com/.

[2] The arduino open source hardware platform. http://arduino.cc/.

[3] R. A. Baratto, J. Nieh, and L. Kim. THINC: A Remote Display Architecture for Thin-Client Computing. Computing Science Technical Report

CUCS-027-04, Department of Computer Science, Columbia University, 2004.

[4] R. A. Baratto, S. Potter, G. Su, and J. Nieh. Mobidesk: mobile virtual desktop computing. In *MobiCom '04: Proceedings of the 10th annual international conference on Mobile computing and networking*, pages 1–15, New York, NY, USA, 2004. ACM Press.

[5] Fraps. http://www.fraps.com/.

[6] M. Kozuch and M. Satyanarayanan. Internet suspend/resume. In *Proceedings of the 4th IEEE Workshop on Mobile Computing Systems and Applications*, 2002.

[7] S. Osman, D. Subhraveti, G. Su, and J. Nieh. The design and implementation of zap: A system for migrating computing environments. In *Proc. 5th USENIX Symposium on Operating Systems Design and Implementation*, pages 361–376, 2002.

[8] M. Samdanis, Y. Kim, and S. H. Lee. Remediation of the wearable space at the intersection of wearable technologies and interactive architecture. In *Proceedings of the Alt-HCI Conference*, pages 1–6, 2012.

[9] M. Samdanis, Y. Kim, and S. H. Lee. The emergence of wearable space: A review and research implications. In *CHI extended abstracts on Human factors in computing systems*, 2013.

[10] M. Satyanarayanan, P. Bahl, R. Caceres, and N. Davies. The Case for VM-based Cloudlets in Mobile Computing. *IEEE Pervasive Computing*, 8(4):14–23, 2009.

[11] Y.-Y. Su and J. Flinn. Slingshot: deploying stateful services in wireless hotspots. In *MobiSys '05: Proceedings of the 3rd international conference on Mobile systems, applications, and services*, pages 79–92, New York, NY, USA, 2005. ACM.

[12] Y. Takeuchi. Synthetic space: inhabiting binaries. In *CHI'12 Extended Abstracts on Human Factors in Computing Systems*, pages 251–260. ACM, 2012.

[13] C. Taylor and J. Pasquale. Improving video performance in vnc under high latency conditions. In *Collaborative Technologies and Systems (CTS), 2010 International Symposium on*, pages 26–35. IEEE, 2010.

[14] C. Taylor and J. Pasquale. Towards a proximal resource-based architecture to support augmented reality applications. In *Virtual Reality Workshop (CMCVR), 2010 Cloud-Mobile Convergence for*, pages 5–9. IEEE, 2010.

[15] C. Taylor and J. Pasquale. Performance aspects of data transfer in a new networked i/o architecture. In *Network Computing and Applications (NCA), 2012 11th IEEE International Symposium on*, pages 155–158. IEEE, 2012.

[16] C. Taylor and J. Pasquale. A remote i/o solution for the cloud. In *Cloud Computing (CLOUD), 2012 IEEE 5th International Conference on*, pages 670–677. IEEE, 2012.

[17] C. Taylor and J. Pasquale. A highly-extensible architecture for networked i/o. In *Computing, Networking and Communications (ICNC), 2013 International Conference on*, pages 867–871. IEEE, 2013.

[18] Tight VNC. http://www.tightvnc.com/.

[19] M. Weiser. The computer for the 21st century. *Scientific american*, 265(3):94–104, 1991.