The Computer as a Dynamic Set of Devices

A thesis submitted in partial satisfaction
of the requirements for the degree
Master of Science in Computer Science

by

Andrew L. Spencer

2016
Mobile, wearable, and ubiquitous computing devices are increasingly replacing personal computers for many everyday tasks. In this thesis, we treat these devices not as individual computers but as part of a larger computing environment and explore the challenges and opportunities presented.

We start by outlining our vision for this environment, focusing in particular on interacting with public devices that do not belong to a single user but are instead provided to the public by businesses and organizations such as restaurants, shopping centers, employers, universities, and governments. Software running in this environment will treat all the currently available devices as a single computer using the unique capabilities of each device to provide an enhanced user experience.

A software architecture is proposed to simplify writing applications that make use of many different devices. We propose a runtime environment to allow HTML and JavaScript applications to execute on a diverse set of devices and develop a concept for virtual components that allows user interfaces to spread across multiple displays without requiring foreknowledge of available devices and their capabilities. For device discovery and communication a JSON encoded protocol is used that operates on top of a number of transport protocols including UDP, TCP, and WebSockets. In addition, a secure boot mechanism is described for providing trusted access to public devices. Lastly, several example applications are provided that make use of this environment and their differences and unique capabilities are discussed.
The thesis of Andrew L. Spencer is approved.

Jens Palsberg

Miodrag Potkonjak

Todd Millstein, Committee Chair

University of California, Los Angeles

2016
# Table of Contents

1 **Introduction** ............................................................................................................. 1

2 **Environment** ............................................................................................................. 5
   2.1 Specialized Devices ................................................................................................. 5
   2.2 Many Devices .......................................................................................................... 7
      2.2.1 Device Discovery .............................................................................................. 8
      2.2.2 Peer-to-Peer ..................................................................................................... 9
      2.2.3 App Migration ................................................................................................... 9
   2.3 Shared Devices ........................................................................................................ 10
      2.3.1 App Discovery .................................................................................................. 11
      2.3.2 Workspaces ...................................................................................................... 11

3 **Programming** ......................................................................................................... 15
   3.1 Overview and Terminology ..................................................................................... 15
   3.2 Components ........................................................................................................... 16
   3.3 Software Hosts ........................................................................................................ 18
   3.4 Drivers ..................................................................................................................... 19
   3.5 Virtual Components ............................................................................................... 20
   3.6 Generic Interfaces .................................................................................................. 21

4 **Networking** ............................................................................................................. 22
   4.1 Transport Protocols ............................................................................................... 23
   4.2 Broadcast Proxies ................................................................................................. 24
   4.3 Message Encoding ................................................................................................. 25
LIST OF FIGURES

2.1  Workspace Manager View  .................................................. 14

5.1  Client Side Encryption ..................................................... 32
5.2  Cryptographic Signing ...................................................... 35

6.1  Media Player ................................................................. 42
6.2  Chat Application ............................................................ 44
6.3  Caching and Duplication Policies .......................................... 48
6.4  File Manager ................................................................. 49
6.5  Text Editor Open Dialog ................................................... 50
CHAPTER 1

Introduction

Many people have proposed that the personal computer era is nearing an end. Indeed, mobile devices such as smart phones and tablets have already replaced stand alone PCs for many applications such as entertainment and reading. This thesis anticipates a future technological environment consisting of many small customized devices and explores some problems and opportunities when transitioning to such an environment.

We begin by looking at some of these changes from a physical perspective, looking at the types of devices, the number of devices, and the ownership of the devices. We then look at how our current software development practices relate to these new types of devices and determine how software design can change to simplify programming and improve the user experience.

Specialized devices that use a variety of different sizes and form factors will become more prevalent. Mobile devices already fall into this category; however, we will also see many other types of special purpose devices such as in-vehicle entertainment systems, networked environmental sensors, and smart projectors and media centers.

Currently software is developed separately for each type of device. For instance, iPhone, Android, and Windows Phone all have separate development environments. This works for a small number of devices, but as more and more specialized devices are produced, the number of development environments will increase significantly.

Instead, we propose a common runtime environment that can execute on many different types of hardware. Generic software and hardware “capabilities” can then be defined so that applications are not dependent on specific hardware devices.
**Multiple devices** will work together to accomplish a given task. Many different specialized devices will be used together instead of relying on a single general purpose computer. A current example of this is a media center that contains a display, speakers, game console, and remote control. As devices become more powerful, each of these components can be used as part of the media center or as stand alone devices. In addition, we will see examples of collaboration between other devices such as a smartphone acting as a remote control or a tablet PC displaying a channel guide and movie reviews.

Software is currently designed for a single device. When multiple devices interact, the software for each type of device is written separately and then installed and run independently on each device. As the number of devices and device types increases this becomes increasingly problematic. Each software application will need to be aware of every other possible application or device type that it will interact with.

Instead, we propose a *one-app model* that treats multiple devices as a single computer rather than separately. In this way, apps can migrate between different devices as the user moves about the environment and multiple devices can be used together for the same task. This allows application developers to write a single application that makes use of many devices simultaneously instead of writing separate applications for each. This can be thought of as writing a single media center application that uses the screen, speakers, and remote control as *components* within a single piece of software.

**Shared devices** will become common and will lead to a reduced dependence on one’s own personal devices. Many of these devices will be provided to the users by businesses and organizations such as restaurants, shopping centers, employers, universities, and governments. An example of this would be an airline which includes a personal entertainment device built into each seat. Other examples of shared devices would be media centers shared among a household and projectors and computing lab facilities provided by employers, universities, and libraries.

Since most users own their own devices, they implicitly trust in them. However, when using shared devices extra attention must be given to security since they are easier
to compromise and may contain malicious software designed to steal the personal information of anyone accessing them.

Many public services, such as websites, use SSL or TLS for secure communication. Software and applications can also be verified using cryptographic signing as is done with many mobile apps and during the UEFI boot process. A similar hardware assisted boot mechanism can be used to verify the integrity of public devices. Additionally, cross device communication can be encrypted and “workspaces” can be used to compartmentalize different types of data.

This thesis proposes a computing environment that allows a single application to spread across multiple devices rather than executing on a single device. In doing so, software development is simplified and the user experience can be enhanced.

For the purpose of developing and analyzing our proposed changes to the software development environment, we have chosen several canonical examples of applications that are challenging to implement using current practices in order to highlight the advantages of our proposed methods.

To support these example applications, an HTML5 and JavaScript framework is developed that runs in either a web browser or in a special purpose Android application. The framework uses UDP, TCP, and WebSockets for communication, depending on the runtime environment, and relies on a separate broadcast proxy to relay messages between devices. Applications residing on the same subnet can also communicate directly using UDP broadcast or multicast. Broadcast communication also serves as a simple method of application and device discovery.

The first application is a media player that includes a display, play list, and remote control. The primary purpose of this application is to test development of software using the one-app model. It uses virtual components to move parts of the user interface between different devices when they become available. For example, when a smartphone is detected, the remote control features will automatically be displayed on the touchscreen.

The second application is a communication application that focuses on sharing informa-
tion between different users. This tests the limits of the one-app model by incorporating multiple devices owned by different users.

Lastly, a file manager application focuses on sharing and synchronizing information between devices belonging to the same user. This serves as a useful application on its own but is also needed to enable other applications to work successfully.

The remainder of this thesis is organized as follows: Chapter 2 introduces a hypothetical future computing environment consisting of many different devices working together and being shared among different users. Chapter 3 proposes a software architecture that simplifies the creation of applications designed to run simultaneously on many devices. Chapter 4 describes networking protocols and APIs used for communication. Chapter 5 discusses security concerns when using untrusted public devices. Chapter 6 provides several sample applications and evaluates the benefits of our programming model. Chapter 7 concludes this thesis.
CHAPTER 2

Environment

In this chapter, we take a closer look at a multi-device computing environment from a user’s standpoint. The chapter begins by looking at a number of specialized device types that are becoming available and anticipates the ways that these devices will be used. It then focuses on how multiple devices will interact, including communication and device discovery. Lastly, a concept for workspaces is introduced that allows multiple devices to work together and share information in a controlled way.

2.1 Specialized Devices

Current computing systems largely revolve around personal computers and mobile devices. For most users, these devices serve different purposes and tend to complement each other [5]. If we hope to design a computing system for many different devices, we need to begin by looking at the types of devices that will be available and the specific uses for each of them.

Traditionally scale has played an important role in classifying devices into different categories, such as tabs, pads, and boards [9]. In addition, a large variety of special purpose computing platforms, such as entertainment centers, in-vehicle entertainment systems, and home automation systems, are becoming available.

A few of these device types are described in more detail below, along with how they are envisioned to fit into a many-device computing environment:

Workstations including laptop and desktop computers are likely to remain similar to existing personal computers for some time to come. The main difference is that they will become more connected to other devices within the computing environment. However,
it is expected that many leisure activities performed on workstations, such as listening
to music, watching videos, or reading the news, will be performed on different devices
specifically suited to those activities. Workstations will typically contain more input
devices such as keyboards and mice.

**Mobile devices** such as tablets, smart phones, and smart watches will continue to be used. However, it is expected that the reliance on one’s own personal devices will decline as computing platforms are built into more every day systems. For example, in-vehicle entertainment systems have largely replaced portable TVs, DVD players, and laptops that were previously used while traveling.

**Entertainment centers** and television sets will take on a larger role as they become capable of performing more complex tasks, such as browsing websites and reading email, that have traditionally been done on workstations.

**Smart projectors** will largely replace existing projectors. The distinction being that a smart projector will be a computing platform in itself rather than simply a raster display device. For example, a presentation residing on a smartphone can be sent directly to the projector for viewing without needing a separate laptop or workstation connected via a video cable.

**Whiteboard** type displays will work similar to smart projectors except they will also provide an input mechanism that can be used by a number of participants.

**Home automation systems** will find increasingly widespread use. For the most part these will be simple devices that either provide a user interface (such as a light switch) or perform some type of environment control (such as a light). However, treating these as separate devices fits nicely into the computing environment.

**Shared systems** are an example of a new class of devices that will be built into everyday objects, such as vehicles, office desks, living room walls, or restaurant tables. The defining feature of many of these systems is that they are shared between many different users. Unlike today’s computing platforms that are typically owned and used regularly
by a single person, these systems will be available to anyone within the vicinity and, in many cases, will only be used a single time.

**ID and data fobs** will facilitate the use of shared computing systems by providing information about the current user. These devices will behave similar to single sign-on systems but provide additional capabilities such as data storage, and synchronization. An important feature of these devices, as described in detail later on, is to ensure that the shared devices being used have not been compromised or tampered with.

Initially these capabilities may be integrated into other portable devices such as cell phones and smart watches. However, dedicated devices may provide benefits in terms of battery life and smaller weight and size since they will rely on other devices for input and output.

**Storage and compute** systems will provide long term data archival and high performance processing. These systems may either reside in the home, an office, or in a datacenter.

### 2.2 Many Devices

The device types discussed so far are often small or provide limited functionality on their own. As such, we can easily recognize the need for communication between devices. A simple example of this would be developing slides on a workstation, transferring them to a smartphone, displaying them on a smart projector, and receiving comments and discussion notes via a smart whiteboard.

We will now discuss some desirable characteristics for cross device communications. First, a simple discovery mechanism is needed so that the available devices can be used. Second, applications should continue to function if network connectivity is lost; for example, while traveling in a plane or while located in a building with poor wireless reception. Lastly, users should be able to transition easily between devices. This means that they should not have to log out and log back in when switching to a new device. Instead, the software should be capable of using multiple devices simultaneously and transitioning from one device to
another without interruption.

2.2.1 Device Discovery

A key distinction to this new computing environment is the ability of the user and the software to make use of any and all computing devices available at any given time. This requires a device discovery mechanism where nearby devices can be quickly enabled and integrated into the software environment. Geolocation may eventually play a role in this process but it is not expected to be necessary for a successful system. Initially, devices can be “discovered” and “paired” via simpler proximity methods or through manual user selection mechanisms.

One could envision a home computing system consisting of displays and input devices available in every room. Prior research projects have focused on incorporating ways to locate the user and automatically switch on or enable the appropriate displays for their location. Additional studies have proposed knowing the relative locations of various devices to enable “hyperdragging” from one device to another [3,7]. Mechanisms for enabling and configuring inaccessible devices such as billboards have been proposed as well [6]. While these types of interactions are certainly beneficial they are not perceived to be a prerequisite for a successful many-device computing environment.

A simpler computing environment could work without location sensing, while still allowing users to make use of many devices. The simplest and most obvious mechanism would be to have the user turn on the devices they want to use. In many cases this primitive “discovery” mechanism would be acceptable. These types of simple discovery mechanisms, while not perfect, are sufficient for the early versions of a many-device computing environment.

Later, once applications can easily use many devices, advanced discovery mechanisms and geolocation between devices can be added as application level features. This approach of adding discovery and geolocation second is also beneficial because multiple algorithms can be implemented in different applications by different research groups while using the same basic infrastructure for many-device computing. Additionally, application level discovery and
pairing will provide more flexibility to the application developers. For example, a banking application may have more stringent requirements when encountering a new device than a entertainment or multimedia application.

For the purposes of experimentation, a simple discovery mechanism is used whereby any connected devices are automatically available for use. This is implemented using UDP broadcast and multicast communication.

2.2.2 Peer-to-Peer

We propose an environment consisting of a number of “smart” devices running software on their own. This contrasts with existing research and protocols such as VNC that provide “dumb” devices that simply provide input and output functionality to a centralized server or servers [4,10].

A centralized server would be beneficial by providing a more traditional programming environment and would simplify data and control synchronization. However, it is expected that a decentralized system will be more flexible in the long run and will provide a more robust platform that excludes the possibility of a single failure point. In addition, decentralize systems avoid the problems associated with a possible lack of network connectivity or the problem of being tied to a single service provider.

2.2.3 App Migration

An important feature for our software is the ability to “migrate” between devices [8]. Once again we will use the example of creating and sharing presentation slides. When the user leaves their workstation not only are the slides copied to a mobile device, but the presentation software is copied as well. Later, when they begin the presentation, the software is copied from their mobile device to the smart projector and smart whiteboard. By copying the software along with the slides, a more consistent user interface is created for all the devices. This also avoids the issue of incompatible software versions, such as unsupported audio and video codecs on the projector.
We could potentially treat the projector and whiteboard as display devices and simply show the user interface on those devices. However, a very high bandwidth and low latency connection would be needed to display videos or provide interactive content. These limitations are also inherent to existing remote desktop software such as VNC.

A common execution environment is needed to support app migration. This allows software written for one device to execute on other devices and will be discussed in more detail in Section 3.3.

2.3 Shared Devices

Now that we have discussed specialized devices and communication between many devices, we will focus on one particular variant of these: shared devices. In the previous example of presentation software, the user’s workstation and mobile device are likely personal devices. However, the smart projector and smart whiteboard are examples of shared devices because they are used by many different people. Additionally, the presenter may only use them for a single conference and never connect to them again.

The concept of shared devices differs significantly from many of our existing computing systems. Luckily, if a working discovery mechanism is available, these share devices can be treated similar to any other device in a multi-device computing environment. However, we can see a few problems and opportunities when interacting with shared computing resources.

First, users may occasionally encounter new types of devices and new situations that they have not seen before. How does the user then interact with these new devices? To solve this, we propose using *application discovery* in addition to device discovery.

The issue of security should also be addressed. Users will implicitly trust their own devices more than a public device that can be accessed, and tampered with, by anyone. We propose a concept of workspaces to help prevent unauthorized access to data. Chapter 5 will focus on security in greater detail and will introduce a method of authenticating and authorizing devices.
2.3.1 App Discovery

Once computing devices are built into many everyday devices it is likely that software will begin to be distributed with those devices as well. In our presentations software example, the smart whiteboard will come with built-in software that can be “discovered” by nearby users. In this way, instead of copying software from a mobile device to the whiteboard, the software is already available. In addition, the software may be saved back to their mobile device and used for other presentations as well. This provides a solution when encountering new devices.

We also consider new situations. One could also think of an airline that distributes software that lets users track their flight, order drinks, and suggests activities and restaurants at their destination. Many of these features are already provided on websites available to WiFi connected devices. In any case, software, in addition to devices, will become available as the user moves through the environment.

Application discovery and device discovery can be thought of as special cases of service discovery [2]. By providing applications along with services, it ensures that a user entering a new environment will have a way to interact with the service without making assumptions about the capabilities of the devices they bring with them.

An interesting extension to this environment would be for certain devices to serve as repositories for software, similar to an app store, but focused on specific locations. For instance, several applications could be available in a repository at a particular tourist location where users can rate them or upload new applications.

2.3.2 Workspaces

Traditionally, processes and devices implicitly provide for data compartmentalization. For example, when a photo is taken from a cell phone it remains on the phone until it is explicitly shared or copied to another device. Similarly, data contained in memory within an application is inaccessible to other processes without the use of shared memory or some other form of inter-process communication. This is beneficial in that it provides control over how data
is shared and accessed.

In a many-device computing environment, data must be shared both between processes and between physical devices. So far, we have assumed that all our software will communicate equally. However, it would be beneficial to retain some of the data compartmentalization abilities provided by traditional computers. This can be accomplished using workspaces that limit which devices and applications can access certain pieces of data.

Using the above example, when a photo is taken it will be automatically assigned to one or more initial workspaces. Often the initial workspace will be inherited from the application creating the data. This mimics the behavior of traditional computers where files created on a device are only accessible to other software running on that device, or in this case, running in the same workspace. Similarly, the file must be explicitly shared or uploaded in order to be accessible outside its original workspace.

In some cases, we will want to share our data with other users and other devices. For example, a smart projector in a lecture hall will need access to the presentation slides. This can be accomplished in several ways, the slides can be added to the projector’s workspace, the projector can be added to the presenter’s workspace, or a new workspace can be created containing only the projector and slides.

Workspaces can be thought of as a form of “tagging” that restricts access to data. However, they do not provide for multilevel security and there is no concept of “higher” or “lower” privilege levels. Similarly, no constraints are placed on dataflow between workspaces. That is, an application residing in two workspaces may act as a bridge between those workspaces. While more rigorous security policies may be desired for some uses, workspaces are intended to provide similar forms of data compartmentalization that are implicit to existing computers without creating extra burden on the users.

Many different types of workspaces are possible and the differences are primarily distinguished by how they are used. Some example uses include:

**Public workspaces** allow any users and applications to communicate amongst each other.

This provides many important abilities such as application and device discovery. In
most cases, public workspaces will be tied to a particular geographic location and can be thought of as local broadcast interfaces. They may make use of existing broadcast networks such as UDP broadcast over WiFi for communication. Types of communication using public workspaces would include weather reports, road conditions, sales and advertisements, etc.

**Shared workspaces** are similar to public workspaces but would be specifically joined for some purpose. Shared workspaces could be used for communication about a specific type of event or among a related group of people. An employer may provide a shared workspace for all their employees or a home network may be implemented as a shared workspace.

**Private workspaces**, discussed at the start of this section, are defined by the user and are the primary type of workspace visible to the user. By default, every user will reside in at least one private workspace. Communication within private workspaces would include personal files, emails, text communications, and audio and video.

For many users, multiple private workspaces may be used. A frequent example of this may be “work” and “home” workspaces that provide separation of personal and work related files and data. Other private workspaces may include workspaces for different security levels or different types of data such as personal communications and entertainment files.

**Application workspaces** would be used for intra-application communication. In many cases, custom messages between software components would not need to be visible outside the application and would therefore use an application specific workspace. This provides an alternative to per-process address spaces that are used on traditional computers.

**Device workspaces** would provide status information about specific devices. This includes task lists, battery status, security information, and other device specific data. The use of these workspaces should be limited to enable software applications to extend
Both applications and devices may reside in and make use of many different workspaces. As mentioned, both will have application or device specific workspaces but will likely use various private, public, and shared workspaces as well. This is inherently a many-to-many relationship and workspace management software will provide both workspace, device, and application centric views of the environment as shown in Figure 2.1.

Communication between devices within a workspace should be strongly encrypted to ensure security and prevent information leakage. While encryption is the primary method of security, it is also beneficial to limit excess communication both for the purpose of network efficiency and to further improve security. Mechanisms to facilitate joining and leaving workspaces may be used for this. For example, private workspaces may make heavy use of loopback interfaces or packet switched networks while public workspaces will likely make use of broadcast or multicast interfaces.
CHAPTER 3

Programming

Chapter 2 discussed a many-device computing environment from a user’s perspective by looking at the types of devices, how they will operate together, and different tasks they will be used for. We now turn our attention to programming these devices in a way that supports our desired user interactions. Two particular challenges are focused on when developing software for many devices.

First, as many devices begin to be used together, the application complexity will increase significantly. Currently, each application running on a particular device needs code to interface with every other device with which it will interact. Instead, we propose treating all the available devices as a single computing system and writing software that can span across multiple devices simultaneously. Virtual components are then used to help applications developers accomplish this.

Second, as the number of different specialized device types increases, problems will be encountered developing separate applications for each of them. Currently, separate applications are developed for desktop computers, tablets, and smartphones. In addition, many embedded devices have special purpose software that cannot be changed. We propose using software hosts to provide a common execution environment so that applications can run on many different types of devices.

3.1 Overview and Terminology

We start by defining some terminology that will be used throughout the remainder of this chapter. The computing environment consists of a number of physical devices, such as laptop
computers and cell phones. Each device has one or more capabilities that are used by the application or applications running on the device. For example, a laptop would provide display, input, and storage capabilities. Many of these capabilities are provided by drivers that interface between the underlying hardware and the application.

Applications running in this environment are comprised of a number of components each providing additional capabilities to the rest of the application. A component is distinguished by providing some specific functionality to the user. For example, a dial pad on a cell phone or a file chooser on a laptop. Note that the user interface for components such as a dial pad may be implemented either as hardware or as software. The key distinction between applications and components is that applications will be spread across a number of devices whereas a component will typically be limited to a single device.

Comparing this to an existing example of a computing system, a home entertainment system, a number of similarities can be seen. The “application” in this case is the entertainment system as a whole. Our example entertainment system consists of a number of different physical devices, including a display, the speakers, a remote control, and a set-top media player. Components in this system would consist of the remote control, the audio/video decoder, the channel listing, etc.

### 3.2 Components

Applications running in a many-device computing environment should be able to spread across and use multiple devices simultaneously. Components provide a way of accomplishing this by breaking the software into stand alone pieces that can execute on different physical devices and communicate amongst each other. Components are different than separate applications in that they are provided together in a bundle. Furthermore, they are managed by a runtime environment that decides which components run on which devices. This allows the application developer to write software as if it were a single application rather than one or more communicating applications.

Each component provides additional capabilities to the rest of the software. It is possible,
and in fact likely, that more than one component will provide the same capability. A common example of this is the volume controls on many cell phones. There may be a set of physical buttons for adjusting the volume located on the sides of the phone and a second volume control on the screen. In this particular case, the physical buttons are said to be a hardware component while the on screen volume controls are a firmware or software component. These different types of components are described below:

**Hardware** components are distinguished by having physical buttons or controls specially designed for that component. Volume controls on a phone or laptop, the buttons on a remote control, and the climate control buttons and switches in a vehicle are all examples of physical components. In many cases using a physical component in lieu of a software component can be beneficial to the end user. A physical component may be simpler or more intuitive to use or may provide better tactile feedback if the user is focused on another task such as driving.

**Software** components are provided by the application and make use of generic capabilities such as an LCD display, a microphone, or speakers. The advantage to software components is that they can be used on any number of devices; however, they may not be as intuitive or as customized as a hardware or firmware component. The majority of each application will be implemented as software components. In some cases a software component being used by an application may be provided by a third party. Common examples of this are save-as dialogs or the fore mentioned volume sliders. Libraries of generic components can be developed and used by multiple applications. These may be provided either by the operating system or a third party software vendor.

**Firmware** components represent a cross between hardware components and software components. To the user, firmware components appear similar to software components in that they make use of generic hardware features such as touchscreen displays. However, they are designed specifically for a single piece of hardware and cannot be transferred or executed on other devices. Firmware components are loaded onto a device by the manufacturer and may be implemented in a lower level language and use device specific
features and APIs. An example of a firmware component might be a task switcher or process manager that is used to control other applications and software components.

Application software should be designed to interact with the capabilities provided by a component in a way that allows the implementation to change. Applications can then make use of newer or customized devices without needing to update the software to add support each new component. For example, a media player may provide a default software component to change to the next or previous track, but when the user is in a vehicle, they may wish to use the physical buttons built into the dashboard instead. If the application provides a default volume control component but interacts with it through a generic interface then this transition becomes seamless.

3.3 Software Hosts

A software host is a special type of component designed to run other software components. These provide a common runtime environment that allows applications and components to run on a variety of different device types. Typically, these will be implemented as a firmware component and provided by the device. This thesis focuses on a specific type of software host, a Web Host, that uses HTML5 and JavaScript. However, several other types of software hosts are possible as well:

Web hosts provide an execution environment for HTML and JavaScript based applications and components. This has the advantages of supporting a wide range of devices and is a widely used platform for developing software [1].

Android hosts would allow execution of native Android apps. This provides performance improvements over web hosts and allows for direct access to the Android runtime.

Java hosts are similar to Android hosts but would execute standard Java SE binaries, which are typically packaged as JAR files.

Native hosts would execute machine code for the target processor. This provides the best
performance and is the most flexible. However, they have two disadvantages in that they provides fewer high level abstractions and executable code must be produced for multiple machine architectures in order to ensure portability.

For many devices, such as tablets and smartphones, the initial software loaded by the manufacturer will consist of a software host, a set of drivers providing hardware access, and some firmware control components such as a task manager.

Software hosts should be provided by any general purpose computing device. However, simple devices such as light switches or remote controls may not benefit from this because they do not include configurable input and output hardware such as LCD displays and touchscreens.

Transitionally, software hosts can be implemented as a layer on top of an existing system. For example, as a layer on top of a web browser using JavaScript and HTML5, as an Android application, or as a background process running on a desktop computer.

3.4 Drivers

In addition to software hosts, devices will provide a number of different drivers. Drivers are implemented as firmware components and provide access to the hardware features of a device. These may interface directly with the hardware or may be implemented as a translation layer between an underlying operating system, such as Windows, Android, or a web browser, and the rest of the components in the system. Like software hosts, common driver interfaces allow applications to run on different types of devices. Some common types of drivers include:

Speaker, microphone, display, and camera drivers allow interacting with the audio/video input and output capabilities of the computer. Camera and video drivers would provide both an instantaneous interface for photos and a streaming interface for video. Audio drivers would primarily provide streaming interfaces.

Pointer and keyboard drivers would be used for receiving various types of human input.
By providing these as a first-class networked driver interface it is easy to decouple the input
devices from the software and the display devices. Unlike today, a user could sit down at a
desk and use a keyboard that automatically interfaces to software running on a smartphone
and a displayed on a projector.

Storage drivers would provide the lowest level filesystem interface for applications. The
abstractions provided by a storage driver may not necessarily correspond to the abstractions
that the user will see. For example, a storage driver may provide a unique identifier for
each piece of data, such as a hash or a file path. However, a storage application would
provide a more user friendly abstraction that supports tagging and searching for pieces of
data. In addition, the storage driver would only keep a single instance of the data on a single
device; whereas, a storage application may handle synchronization and data duplication for
the purpose of backups and offline use.

Network drivers would provide an interface to legacy applications and network protocols.
For example, an email application could implement the IMAP and SMTP protocols using a
TCP driver. It is important to note that network drivers are not needed for communication
between components. For example, a small device connected via a wireless link such as
zigbee or bluetooth could potentially communicate with TCP services using a TCP driver
from another device. Additionally, it is possible that no TCP drivers are provided even
though cross device communication uses UDP or TCP underneath.

3.5 Virtual Components

Previously we have referred to multiple devices as a single computing environment. A single
application can then be written that runs simultaneously on all these devices. This is referred
to this as a one-app model in contrast to developing a separate application for each device.

One way of accomplishing this is to use virtual components. These components are
elements in a graphical interface that contain control and user interface code. A virtual
component is a wrapper around a component that allows it to be displayed on one or more
different devices.
Similar to interfaces in object-oriented programming, virtual components provide a more abstract programming interface. We can say that an interface abstracts the implementation of a component; whereas, a virtual component goes one step further and abstracts the instantiation of a component as well. That is, a virtual component may be instantiated on a local device or on another device, or there may even be multiple different instances of the component running on several devices at the same time.

Using virtual components, the programmer is able to write software for an application that is conceptually running on a single device, while the underlying runtime system handles mapping and distributing the components to the available devices in the best possible manner. This is analogous to the use of “tabs” in existing applications such as web browsers. In this case, the programmer creates a number of tabs in the application and the graphical toolkit, in conjunction with the operating system, places the tabs into one or more windows and displays the windows on the screen or screens.

### 3.6 Generic Interfaces

As mentioned above, virtual components provide an instantiation abstraction. However, it is beneficial to combine this with traditional interfaces as well. This allows different vendors to provide custom or improved implementations of interfaces that can be used transparently by the software.

One could think of a navigation and routing application. One of the components in such an application would be a map display component. By using a virtual component with a well defined interface, it is possible for different implementations of the map to be used simultaneously. For example, when looking up directions at home a software map component will be used on a laptop. Later, the same application can continue to function using the map and navigation software built into a vehicle.
CHAPTER 4

Networking

So far we have discussed programming for many devices from a conceptual standpoint and have largely ignored implementation and communication details. We now turn our attention to these by describing a simple communication mechanism that has been used to implement several sample applications.

The network architecture consists of three parts: a transport protocol, a message protocol, and higher level programming APIs. The transport protocol, such as UDP, TCP, or WebSockets, is used to transmit data between devices. The data is formatted as one or more JSON encoded messages. Higher level APIs can then be designed to simplify the message processing.

First, a transport protocol must be chosen. The primary requirement is to facilitate device discovery and peer-to-peer communication. For implementation purposes it should also work well with existing devices and operating systems. Unfortunately, no single transport protocol was found that supports all these needs. Instead, a number of different protocols connected together via a broadcast proxy can be used. This proxy serves to relay messages from unicast protocols such as TCP and WebSockets to other clients and also to any locally available broadcast interfaces such as UDP broadcast and multicast.

The second choice, the JSON encoded message protocol, was driven by several factors. First, the messages should be easily parsable using JavaScript, which was chosen as a common development language used for prototyping. From a design standpoint, the messages should be self describing and easily added and extended without needing a centralized list of message IDs and message formats.

Lastly, a number of APIs are defined that can be used to transmit data. At the lowest
level a broadcast function transmits messages and a listen function registers a callback that receives messages. These two primitive functions are then used to create a driver API and virtual components that help simplify application development.

4.1 Transport Protocols

Ideally, a single transport protocol would be used for all communication. However, for the purposes of developing sample applications it was found that no single protocol provided all the desired features. A number of different protocols were evaluated instead and used together in a prototype environment.

HTML and JavaScript were chosen as a development platform and our evaluation of transport protocols focused on their suitability both for use in a web-based environment and for their ability to support many-device computing as described in the previous chapters.

In the future it may be possible to switch to a single protocol, but for now a combination of the following protocols was used:

**UDP** broadcast packets are the primary communication mechanism. This provides a simple device discovery mechanism and facilitates peer-to-peer communication between a large number of nearby devices. Multicast communication can also be used for applications that are not connected to the same subnet. Another technical advantage of multicast over broadcast is that devices can avoid receiving copies of packets they transmit by disabling multicast loopback.

The main disadvantage to UDP is that it cannot be used directly in our prototype environment because most web-browsers do not directly support it without extensions or special development permissions.

**TCP** can be used as a link-to-link protocol. Normally, an advantage of TCP is that it provides reliability. However, in our environment we assume that devices will frequently connect and disconnect as the user moves about. Due to this, reliability in the form of TCP retransmission becomes less beneficial, especially for low latency interactive
applications. Additionally, TCP does not support broadcast messages and, like UDP, is not supported in web-based applications.

Despite these disadvantages, TCP can still be useful as a relay between a device and a broadcast proxy, as described in Section 4.2. In addition, TCP has certain practical advantages. For example, Android requires extra permissions for multicast communication. Broadcast and multicast receive may also be disabled on low power devices to prevent excess CPU wakeups.

**WebSockets** is used as the default transport mechanism for HTML5 applications. WebSockets are beneficial because the entire application stack can be implemented in HTML and JavaScript without the need for plugins [1]. Unfortunately, WebSockets supports neither broadcast, multicast, nor peer-to-peer communication. As such, WebSockets must also be used in conjunction with a broadcast proxy. Some future protocols, such as WebRTC, may eventually allow for direct peer-to-peer communication between web browsers; however, these are not yet widely supported.

**HTTP** is not used except in the initial setup phase of WebSockets. The main advantage of HTTP is its wide support; however, this is not seen as a significant enough advantage because most recent browsers support WebSockets as well. In addition, HTTP is designed as a client-server protocol and is therefore less suited for long running peer-to-peer communication.

### 4.2 Broadcast Proxies

As noted above, the application environment relies heavily on broadcast messages for device discovery and cross device communication. However, WebSockets and TCP/IP do not support broadcast. A key enabler for broadcast messaging is an application that serves as a proxy between applications and components using different transport mechanisms. This allows HTML5 applications using WebSockets to communicate with Android applications using broadcast UDP or TCP/IP, for example. The use of a broadcast proxy as a relay
between different protocols is mostly an implementation necessity due to the current state of network protocol support in web browsers.

In addition to relaying messages, the proxy may publish its own capabilities to allow applications to switch between transport protocols. For example, an Android application could initially connect via UDP broadcast, then switch to TCP once a broadcast proxy is discovered. The Android device and other low power devices could then reduce their battery usage while still receiving messages.

4.3 Message Encoding

The JSON format was chosen as the message encoding for our initial implementation. JSON is convenient because it allows for arbitrarily complex messages in a simple human readable format. Other encodings could be used instead, but it is important that messages are self describing so that applications do not rely on a globally defined list of message IDs. This also allows additional fields to be added while maintaining compatibility with existing components and devices.

It is possible that certain messages may use binary encoding as well. This would primarily be an optimization for high throughput applications such as audio and video streaming. Additionally, large file transfers may be less suitable for a message-based protocol and would instead use TCP/IP or some other communication mechanism directly while JSON messages are used for control purposes.

4.4 Capabilities

All devices, applications, and components will generally implement at least one “capabilities” message that is broadcast at a periodic frequency, for example, once a second. The primary purposes of this message is to describe the additional messages that can be sent and received by the component.

A secondary purpose is to serve as a health monitor. Since most users will be moving
about the environment they will likely be connecting and disconnecting from devices frequently. The high rate of the capabilities message ensures that the software can quickly adapt to the changing environment. One point to note is that there are no globally defined “disconnect” messages. The software is expected to monitor the capabilities messages to determine when devices are disconnected or powered off. In more advanced implementations, separate messages may be used for capabilities and connection monitoring in order to reduce network usage.

4.5 Programming APIs

So far we have discussed networking and communication from a message passing standpoint where separate devices send messages to each other. Initially this seems contradictory to our goal of one-app programming that was introduced in Section 3.5. However, message passing can still be used as the underlying communication mechanism as long as higher-level APIs are built on top of it. This is analogous to using a widget toolkit or a scene graph for graphics programming in order to avoid bitmap drawing operations.

Some of the programming APIs used in our sample implementation are described below. Different implementations may use other APIs as well, but it is expected that higher level APIs that focus on programming for many-devices as a single computer will have significant advantages over message-based APIs that treat each device separately.

Broadcast is the lowest level of communication. Our sample broadcast API provides broadcast, listen, and remove functions. Applications will register a listener callback that will receive all broadcast data, the listener can later be deleted using the remove function. Data is sent using the broadcast function.

Despite being a very low-level API, this can still be conceptualized in some ways as one-app programming. First, using listeners and callbacks avoids having to explicitly write a per-device event loop. Additionally, each component can register its own listener function and be developed in an object-oriented style where the messages correspond
to the component’s public methods. Components, treated as objects, can then be serialized and sent to other devices where their listener functions are reconnected.

Drivers are commonly implemented as simple command response style components. As such, it is beneficial for them to easily filter and parse relevant commands. Broadcast listeners are built upon to develop a message parsing domain specific language that more closely matches traditional object-oriented programming.

Message parsing is currently implemented on the Android platform using Java Annotations. The annotations consist of a list of patterns which are matched against incoming messages. If all the patterns match, the message is passed to the handler. Consider the example of a data storage driver that sends and receives the following messages:

Request:  
```
{"read":{"name":"test.txt"}}
```

Response:  
```
{"read":{"name":"test.txt","data":"hello, world"}}
```

The following code would be used to parse the request while ignoring responses by other data storage components.

```java
@Receive("read.name:str", "!read.data")
public void onRead(String name) {
... }
```

Introspection may be used to automatically connect methods and/or function definitions to the corresponding messages. It should be noted, however, that messages are generally broadcast rather than sent to a specific object. In addition, the message filters implemented for drivers provide a means of rejecting certain messages; whereas, message passing and pattern matching in object-oriented and functional languages generally only provides a mechanism to accept messages.

Virtual components are an example of a higher level API that uses message passing internally. These provide a mechanism to move components from one device to another. When a virtual component is created, it is given the constructor for another physical component. Initially the physical component is instantiated on the local device. Later on, if another remote device becomes available, the physical component’s constructor
is transferred to the remote device and instantiated there. Once the remote component is executing, the local component is disabled and hidden. If the remote device is eventually disconnected, the component is reinstantiated on the local device.

Virtual components have been implemented in JavaScript using objects and class serialization. In the following example, the MediaPlayer application defines a concrete RemoteControl component that would include play, pause, next, and previous buttons. The RemoteControl constructor is passed to the Virtual constructor which wraps the remote control in a virtual component. The parent parameter is an HTML Element in the local device where the component is initially instantiated.

```javascript
// Media player application
MediaPlayer = function(parent)
{
    // Remote control component
    RemoteControl = function(parent) { ... }

    // Media player main
    this.remote = Virtual('RemoteControl', RemoteControl, parent)
}
```

The Virtual object initially uses a broadcast listener to discover nearby devices supporting the WebHost capability. Once a device is discovered, the component code is broadcast, using a UUID to specify the particular WebHost that should be used. The virtual component then monitors the status of the WebHost’s capabilities message to determine if it eventually is disconnected.

This is one example of a virtual component API. In more advanced systems, a complex runtime system could be used to position and orient objects on multiple displays. Preferences could also be stored to remember what devices a particular user prefers.

Toolkits can be created containing classes such as the previously mentioned Virtual class. In addition, widgets such as file choosers, color pickers, and volume controls can be created and reused. This is analogous to graphical applications that rely on graphical toolkits and widget sets rather than raster or vector graphics operations. Ideally,
toolkits will provide the majority of the networking code and simplify application development.
CHAPTER 5

Security

Section 2.3 looked at devices that are shared among many different users. The smart projector and smart whiteboard in the presentation software example are two types of shared devices. Other examples include seat-back screens in airlines, public library computers, and shared lab computers. We predict these types of shared devices will become more common as computing devices are provided by businesses such as restaurants and shopping centers.

Unlike personal computers, these devices can be accessed by many different people. Malicious users could potentially install key loggers and other software designed to steal passwords and sensitive personal information. The concept of treating many devices as a single computer further complicates security because shared devices could potentially have the same access to our data as our personal devices.

We begin by looking at how existing security mechanisms can be used to help secure our devices. At a minimum, encryption should be used when communicating between devices. We also demonstrate how workspaces can be used to protect sensitive information, such as encryption keys and passwords, on certain highly trusted devices.

Additionally, some assurance should be provided that the devices being used have not been compromised by an attacker. This is referred to as a hardware centric security perspective and a framework is proposed to verify new shared devices as they are discovered.

5.1 Limiting Data Access

Section 2.3.2 introduced workspaces and mentioned that they can be used to provide limited data access to certain devices. In the presentation software example, the smart projector
was only given access to published academic files. Other examples would be giving an in-vehicle entertainment system access to multimedia files but not financial data, or only allowing access to business documents by company computers.

Workspaces themselves do not provide for multilevel security, but they can still be used as part of a policy that implements different levels of trust. For example, three workspaces could be created and named “low-trust”, “medium-trust”, and “high-trust”. Alternately, more descriptive names such as “music”, “letters”, and “banking” could be used. The specific policies used may be different depending on the user and an analysis of different policies is beyond the scope of this work. However, one such policy is described below to demonstrate how multiple levels of trust can be used in a many-device computing environment.

ID and data fobs will serve as the most trusted devices since they are owned by a particular user and are carried by the user most of the time. An important purpose of these devices will be storing personal information and encryption keys that are used by the rest of the devices in the computing environment.

Cloud-based storage servers may be assigned a low trust level because they are beyond the user’s control and may be considered valuable targets for attackers. As such, only encrypted information should be stored on these services. Compute services, however, must have a higher trust level because they will process unencrypted information. To accomplish this, the user can either rely on a more secure data center or use a personally owned device, such as a workstation, for some of their computational needs.

Lastly, input and output devices have medium trust levels. This is due to the fact that they are displaying or receiving a small amount of unencrypted data. Unlike a compute node, the input and output device only needs to display the aggregated results of the computations. For example, a file search application would need access to all the available files; however, the display device only needs access to the search results.

Figure 5.1 shows an example of using all these devices together. Information from the storage driver can be decrypted either on the key fob or on the compute node. In the latter case, the compute node will only be provided with the decryption keys that are currently
Figure 5.1: Client side encryption with many devices. The storage driver is not trusted and receives no access to the user’s data while the display device is slightly trusted and only receives a small amount of relevant data. The key fob is fully trusted while the compute node only receives the crypto keys that are currently needed. Software and algorithms can then be run on the decrypted data and the results can then be shown on a display device. In the future, technologies such as homomorphic encryption may further increase security by allowing the computations to be performed without providing direct access to the unencrypted data.

5.2 Hardware Centric Security

Limiting data access can help mitigate data breaches if a compromised device is inadvertently used. However, an attempt should still be made to verify that the devices being using are secure. In addition, manually assigning trust levels to every device will become increasingly burdensome as more and more shared devices become available. We attempt to solve these problems by developing a hardware centric security framework to verify the integrity of the newly discovered devices.
5.2.1 History and Overview

Historically, computer security has focused on the authentication and authorization of users. This predates computing environments and was natural as governments and organizations shifted to a computer-oriented workplace. We continue to see a focus on this for web-based applications with technologies such as OAuth and OpenID. This is referred to as a user-centric security perspective because the security authority is verifying and providing access to users.

With the proliferation of “app stores” and software markets we have seen an explosion in the number of software applications that users are installing on their devices. This leads to a software-centric security perspective where the user is verifying and authorizing software via mechanisms such as public key signing and per application permissions. Anti-virus and security scanning software falls under this category as well.

A third perspective that gains significant importance in a many-device computing environment, and the primary focus of this chapter, is a hardware-centric security perspective. Presently most users implicitly trust the devices they use because they own them. However, this will be insufficient as users transition to using a wide variety of shared devices that they neither own nor trust. While the first two perspectives focus on preventing malicious users and malicious software, we now have to focus also on restricting access to malicious hardware and malicious devices.

5.2.2 Example Scenario

An example of a customer dining at a restaurant will be used to illustrate some of the security concerns present when using shared devices. Suppose the restaurant provides a tablet computer at each table that can be used by any of the customers. Ideally, using this shared device will be as secure as using a personally owned device.

Let’s assume a customer wants to check their email using the tablet. In this example, the customer’s primary concern is that no one else can gain access to their email account due to their use of the shared device. We will focus on the following ways that an attacker
could gain unauthorized access:

- A previous customer, or even the restaurant owner, could have installed malicious software onto the tablet. For example, a keylogger or a screen capture application.

- The operating system or system libraries installed on the tablet may contain vulnerabilities that would allow an attacker access to the users account. This could be either intentional or accidental due to a lack of security updates.

- Malicious hardware components may be included in or attached to the tablet to record the users information. Credit card skimmers and USB key loggers are examples of these types of vulnerabilities.

Before the customer starts using the tablet they will want to verify that the device is not affected by any of the vulnerabilities mentioned above. To do this they will need to check that none of the applications running on the device are malicious or contain vulnerabilities and that all the hardware components were created by known manufacturers. First, an ID for each of the applications and components must be obtained in a secure way. Second, the user must verify that all of the IDs are safe to use.

### 5.2.3 Securely Obtaining IDs

As mentioned above, in order to trust an unknown device, all of its software and hardware components need to be verified. In doing so a unique ID needs to be obtained for each of the components. For applications, this is commonly done by computing a cryptographic hash of the applications contents. For hardware components, an ID can be embedded in the hardware itself.

Once the IDs are computed they are provided to the user connecting to the device. In the previous restaurant example, the customer is assumed to be carrying with them a small ID fob that will receive the list of IDs from the tablet and perform the verification steps on the customer’s behalf.
The difficulty lies in verifying that the IDs provided by the device are accurate. If the device was compromised, it could easily send out a fake list of IDs in order to mimic a secure device. This form of ID spoofing can be prevented as follows. First, a challenge-response mechanism is used to verify that the hardware IDs are correct. Next, the hardware computes a cryptographic hash of the operating system during boot. Lastly, the operating system computes a hash for each of the applications as they are loaded. The security of this method relies on the fact that the software components, which are easily tampered with, must be verified by a trusted lower-level mechanism, as shown in Figure 5.2. This process is described in detail below:

Hardware components must provide and prove the correctness of their own IDs; unlike software, there is no trusted lower level component to rely on. This can be done using a challenge-response mechanism where the user provides a random number, or nonce, to the hardware. The hardware then signs the data using a per-device private key and replies with the signed data and a certificate chain which is accepted or rejected by the user. This approach is similar to existing protocols such as TLS that validate the identity of a web server, or in this case, the hardware.
The private keys stored in the hardware must be protected via anti-tamper mechanisms or trusted execution environments that make it difficult for attackers to recover the key. These technologies are similar to those used by TPM chips and the ARM TrustZone. Different hardware manufacturers may provide varying levels of tamper resistance.

For simplicity, we assume that the hardware ID corresponds to an entire device and that care has been taken by the manufacturer to prevent modifications. For example, by glueing the device together or encasing the circuitry in plastic or resin. Alternately, encryption could be used between components within the device and a separate ID could be provided and verified for each of the components.

Software IDs, including the firmware and operating system IDs, must be verified using the hardware because it is too easy to recover a private key stored in software.

During boot-up, the hardware will compute a cryptographic checksum of the firmware and operating system. This checksum, along with the nonce, is then signed with the hardware private key and provided to the user. The software will provide the same checksum along with a signature from the developer and a certificate chain leading to a trusted certificate authority.

This is similar to the secure boot process used by UEFI except that the hardware does not determine whether the software is trusted. Instead, the checksum and signature are saved and provided to the user when they connect to the device. This avoids the problems associated with storing and updating trusted software keys in every device. In addition, different users may trust different operating system providers.

Applications are verified similarly, except the operating system computes the checksums rather than the hardware. A similar process is widely employed when installing and running software on mobile platforms. Again, the main difference is that the operating system will provide the certificate chain to the user rather than validate it itself.

It should be noted that these verification steps do not need to proceed in the order listed above. It is expected that the operating system will boot and load several applications prior
to interacting with the user. The operating system verification and app verification may be provided to the user at any time; however, the user must not trust the software until the hardware is also verified.

For added security the user may chose to reboot the hardware prior to trusting the operating system. This ensures that any prior applications run by other users have not compromised the security of the operating system. Alternately, the hardware may be able to perform an active scan of the system memory to ensure that the operating system has not been compromised. Another option would be for the hardware to record a monotonic list of applications that have loaded since boot so that a malicious application can be detected even if it compromises the operating system.

Going back to the example of a customer using a tablet at a restaurant, we now show how tampering can be detected under a variety of circumstances:

• Suppose a previous customer has installed an app on the tablet, such as a key logger, to record the user’s information. When the next user attempts to connect to the device the operating system will provide a list of the applications that are currently running. The key logger will be included in this list and can then be rejected by the user as an untrusted application.

In some cases, the application itself might not be malicious but would still be rejected by the user because it provides vulnerabilities or back doors. This would apply to software such as an SSH server that provides remote access to the device.

• Suppose someone has modified the operating system on the tablet to provide a fake list of IDs to the user. There are several possibilities here:

  – If the OS fakes the application IDs only, then the user will detect that the OS ID is unknown or untrusted.

  – If the OS provides a prerecorded OS ID and signature (a replay attack), then the signed nonce will not match what was provided by the user.
– If the OS attempts to sign a trusted OS ID along with the nonce itself, rather than using the hardware signature, the user will detect that the signature was not created with a trusted hardware private key.

• Suppose someone has installed a hardware-based data recorder inside the tablet. Ideally, this will be difficult to do and will result in visible damage due to tamper evident packaging. Additionally, the tablet can provide a list of the peripherals, such as attached USB devices, to detect when new hardware components have been added.

One additional type of attack is a man-in-the-middle attack. In this case, a compromised device may be able to provide valid IDs to the user by relaying the nonce used in the challenge-response mechanism to another uncompromised device. Like TLS, end-to-end encryption may be beneficial in these cases or timing or location information may be used to detect some of these attacks. However, a more thorough analysis of man-in-the-middle attacks is left as future work.

5.2.4 Certificate Authorities

Once the software and hardware has been identified the user must determine whether the device is safe to use. This is similar to the existing process of installing an application on a mobile device from an app store. In this case, an application is signed by the software developer when it is compiled, verified by a third party when it is included in the app store, and finally approved by the user during installation. A similar process can be used when connecting to new devices.

Certificate authorities can also play a role in this process, similar to the existing TLS/HTTPS security framework. The advantage of having separate certificate authorities is that they can perform more intensive security audits than an individual user. For software and applications, this will be very similar to existing certificate authorities. For hardware, additional steps may be performed.

Multiple trust levels may be established by certificate authorities based on the extent of
the verification, similar to Extended Validation Certificates. More importantly, trust levels may be established based on the physical security of the hardware devices and the amount of tamper protection. The amount of validation is expected to be greater for hardware manufacturers than software manufacturers because the validation will be a one time cost during the hardware development process for a new type of device. This cost will likely be small compared to the remainder of the development costs associated with new hardware. For tamper resistance, several trust levels are expected:

- Minimal anti-tamper protection. The hardware verifies software but no attempts are made to protect the hardware private keys.

- Anti-tamper devices are used to protect the hardware private keys.

- Snooping and hardware modification protection is used throughout the device. This limits the ability for components to be added or removed from the device or for information inside the device to be monitored by an outsider.

If hardware security is compromised, revocation lists will be used to disable access to compromised devices. At a minimum, this would include revoking specific hardware keys. For example, if the key to a device is compromised that key can be revoked to prevent software duplication of the key. Furthermore, if a specific device model is found to be vulnerable to certain types of attacks, all devices of that particular model may be revoked. This is expected to be done by using a chain of certificates with, at a minimum, certificates for the certificate authority, the manufacturer, the hardware model, and the individual device. Additionally, certificates for specific off-the-shelf security related components may also be provided.
CHAPTER 6

Applications

Several sample applications are proposed to evaluate the feasibility and usefulness of multi-device computing systems. Among these are 1) a media player for audio and video, 2) a communications application focusing on email, text, and video communication, and 3) a personal file manager.

6.1 Infrastructure

While implementing and experimenting with the following applications, a set of infrastructure code was developed to support a multi-device computing environment as described in this thesis. The goal of the infrastructure code is to simplify application development and allow communication between software components and devices. The primary implementation framework used for the applications is HTML5 and JavaScript. Additionally, Java, Python, and C were used in supporting roles as described below. The infrastructure consists of:

**HTML5** runtime libraries provide the majority of the supporting code for developing HTML and JavaScript applications. This includes a WebSockets implementation of the Broadcast API described in Section 4.5. A WebHost implementation using the broadcast API is also provided and is used for hosting the other applications. This runtime also provides some basic utilities to facilitate testing and debugging. The HTML5 Runtime has been tested in Google Chrome and Firefox.

**Android** is used as an additional runtime environment for mobile devices. A custom Android app was developed to host HTML5 and JavaScript software rather than relying...
on the system provided web browser. Using a custom Android app allows access to the
camera, microphone, and SD card that are not normally accessible using JavaScript.
The Android framework was implemented using Java and provides a WebHost base
on a WebKit WebView. The Broadcast API was implemented using UDP broadcast
so that apps can automatically discover other devices running on the same wireless
network. Additionally, a Driver API is provided as described in Section 4.5.

Broadcast proxies are needed to support communication between components using the
HTML5 runtime and components running in the Android application. The broadcast
proxy was initially implemented in Python and was rewritten in C to be more robust. It
supports communication using WebSockets, TCP, UDP broadcast, and UDP multicast.

An app launcher and an app repository are also provided as part of the infrastructure.
These are implemented as firmware components and are used to initially load HTML5 and
JavaScript applications into a WebHost.

6.2 Media Player

The overall goal throughout this thesis has been to treat multiple devices as a single comput-
ing environment. Entertainment centers are already an example of multi-device computing
systems consisting of separate devices for the display, speakers, game consoles or set top
boxes, and the remote control. In the media player application we attempt to recreate an
entertainment center as a single application designed to run simultaneously on a number of
smart devices.

An advantage of the one-app approach is that the entertainment center can easily be
repurposed simply by running a different application. For example, the same display and
speakers could also be used by a video conferencing or presentation app. Similarly, the
software can be used in other ways as well; a media player application written for an en-
tertainment center should work equally well on a laptop computer or with an in-vehicle
entertainment system. Another benefit is that it is easier to add support for new devices
and features, such as dimming the lights when a movie starts playing.

The hardest challenge is to correctly utilize each of the devices and to distribute the user interface and input mechanisms in a way that provides a more convenient user experience. For example, if multiple display devices are available, such as a projector, a tablet, and a smartphone, the most efficient way to utilize each device should be computed. Additionally, as new devices such as a second smartphone are discovered, the user interface should be rearranged appropriately. Another challenge is designing a visually appealing interface for the software while still allowing it to adapt and run on many different types of devices.

The media player was implemented using virtual components, as described in Section 3.3, for each part of the user interface. Each virtual component has a device preference associated with it. For example, the video player prefers running on a projector while the remote control component will prefer running on a smartphone. When devices are discovered and added to the computing environment the matching components are automatically moved to those devices.
The media player implementation focused on the underlying mechanisms for spreading the application across multiple devices. In the future, more advanced policies for component layout could be created and more attention should be given to designing an appealing user interface.

During implementation and testing of the media player a “device simulator” was also created. This allows for testing of the automatic layout policies using a variety of different devices, as shown in Figure 6.1. When a device is “created” in the simulator it automatically becomes available and matching virtual components are transferred to the device.

6.3 Communication

Communication applications provide an opportunity to experiment with implementing multi-user applications in addition to multi-device applications.

Fundamentally, there are two ways that a multi-user application can be developed. The traditional way would be to implement two separate applications that communicate with each other, possibly through an intermediate server. Alternately, both client applications and the central server can be implemented as a single application. It is assumed that implementing a single application will be more straightforward because the programmer can reason about all parts of the system as a whole.

6.3.1 Text Chat

Text-based chat clients are the simplest communication application to implement. In addition, they provide a good environment for designing a multi-user system as a single application.

The traditional approach would be to write a client application that starts and attempts to connect to another user or to a server. Once connected, it enters an event loop and waits for messages from the server, and replies with messages entered by the user. The development of this type of application consists of only one side of the conversation.
The one-app approach would consist of a single user interface with two chat windows and two entry prompts. The application then waits for input at either of the prompts. Then, when an input message is entered, the application displays the message in both chat windows.

One point of difficulty when using a one-app model is the initial setup. With two applications, both users will login and then one user will connect to the other. With a single application a way is needed to bootstrap the application onto both users devices. This could be done by having a central server on which the application starts. When a user connects to the central server the chat application incorporates their local device and presence information is shared among all the connected devices.

The sample chat application works as follows. When a user wants to enter the chat room, they first connect to an “app repository” which serves as the central server. They then launch the chat application and it is copied to their device. At this point the app repository is no longer needed and communication happens in a peer-to-peer fashion. Figure 6.2 shows the chat application running in a browser and communicating with an Android device. Note that the Android device communicates using UDP broadcast rather than WebSockets so that no connection toolbar is needed in the Android app launcher.

The chat application follows the “one-app model” by using the Broadcast API. When
input is received in one of the entry prompts, a broadcast message is sent to all the chat windows, including those on other users devices. In the future, it may be beneficial to provide introspection so that the message can be displayed by calling a function on the chat window rather than explicitly sent as a broadcast message.

### 6.3.2 Video Conferencing

One possible extension to the chat client would be video conferencing. This expands on multi-user communication by providing additional input devices. For example, the user could conceptually be using separate devices for the camera, microphone, display, and various input devices.

It is worth pointing out that since all the devices are treated as one computer, a video conference should be moveable between devices, such as from a cell phone to a laptop computer without having to hang up or reconnect.

Additional sharing mechanisms, such as screen sharing and local device sharing, may also be desired. For example, the remote user may wish to share a PDF document with all the participants. Traditionally, the files would be sent to the participants and they would each open the document on their own.

A shared computing environment provides unique opportunities in that the PDF application may be shared instead of just the PDF file. In this way, the conference can become much more interactive because all the participants can work collaboratively within the same computing environment. The video conference becomes a shared work environment rather than just an audio and video display.

This is similar in many ways to how online document editing works with all the participants working on the same document at the same time. However, the entire computing environment is inherently capable of such interactions.
6.4 File Manager

The examples so far have assumed that all the devices and software can access all of the user’s files and data. Section 3.4 discussed storage drivers and storage applications. The file manager is an example of a storage application because it provides for synchronization and organization while the driver only provides data storage on a single device.

The most important feature of the file manager is to allow sharing and synchronization as devices are connected and disconnected. Additionally, the file manager should provide a more user friendly way of accessing files when compared to the low-level storage driver. This includes providing search capabilities and file organization features such as tagging and versioning.

The file management implementation consists of three parts; the file manager itself, two storage driver implementations, and a text editor used for testing.

6.4.1 Storage Driver

The storage driver, or backend, provides access to either a local device storage or to a networked backup service such as Dropbox, Google Drive, or a personal network-attached storage server. The driver provides a path-based interface for reading and writing files and listing the contents of directories.

Our implementation includes two storage drivers. One driver is provided as an HTML5 and JavaScript application that serves as an in-memory cache. This is also used for testing synchronization by creating multiple instances of the storage driver with different initial sets of files. The second storage driver was implemented in Android using the Driver API described in Section 4.5. By default, the Android driver reads and writes to a folder on the SD card. Both of these storage drivers provide the same read, write, and list capabilities and are used by the file manager and text editor applications.

One of the challenges when implementing the storage driver is transferring large files. The sample implementation relies on a JSON message protocol for communication between
devices. If this protocol is used, large files should be divided into chunks and transferred piece by piece, something that is not implemented at this time. Alternately, another protocol such as FTP, HTTP, or BitTorrent could be used to transfer the files.

6.4.2 Synchronization

One of the goals of programming for many devices is to be able to work with the available devices even if an internet connection is temporarily unavailable. File synchronization and caching plays an important role in supporting this goal. It should be noted that this form of synchronization should not require complex merging strategies because it is designed to synchronize a single user’s files rather than files being modified by several users simultaneously. If a user is frequently modifying files from two devices independently and then merging them, the underlying problem is that the software is treating the devices separately rather than as part of a single, larger, computing environment.

Duplication and caching are closely related forms of data synchronization. In this context, caching is the process of moving files from the greater computing environment onto a local device for performance improvements. Duplication focuses on maintaining multiple redundant copies of data for reliability or network outages. Typically a caching policy will be defined for each storage backend and will be set to a certain size such as 500MB or 10% of the usable space. As new files are accessed old files may automatically be purged from the cache.

Duplication policies focus on maintaining a complete set of data at all times. As such, no upper limit on file size is given. If the data size exceeds the amount of usable space it should be treated as an error and the user should be notified. Additionally, duplicated files will not be purged except by manual changes in the duplication policy. An example of a caching and duplication policy is shown in Figure 6.3.

Generally these policies would only change when new devices or services are added; however, temporary policies may be useful in some circumstances as well. For instance, before attending a conference all files tagged as “project-a” can be duplicated onto a mobile
device so that they are available while traveling. These types of temporary policies would likely be deleted once the conference has completed.

The existing file manager implementation only supports basic synchronization where all files are automatically duplicated on all storage drivers. This is useful for testing but would not be practical with a large number of files.

The file manager itself is implemented in HTML and JavaScript and provides a view of the synchronization status of each of the users devices as shown in Figure 6.4. Currently the device list is fixed. Each device is initially placed in a “disconnected” state and then moved to a “in-progress” state once it is discovered. Once all the files needed by that device have been copied to it, it is placed in a “synchronized” state. Whenever a new file is created, all the connected devices are moved back to the “in-progress” state until they have been resynchronized.

An example use of this deals with mobile devices. When a new file, such as a photo, is created on a smartphone it is initially saved to the smartphones storage driver. Later when the phone is connected to a home network it is automatically discovered by the file manager running on a laptop or server and the photo is copied to the rest of the storage drivers.
6.4.3 Organization

As mentioned above, the file manager should provide a user friendly way to organize and access data. This includes features such as searching, tagging, and versioning.

**Searching** would provide an easier way for users to find files. This should be handled by the file manager rather than the storage driver or the application using the files. The File Manager may provide its own interface for searching, but it should also allow other applications to query files matching a search string.

**Tagging** is one way of organizing files that seems to fit well with a many-device computing environment. Traditionally, users can separate files on different devices; a work computer will have business files, a media center may have audio and video recording, or a laptop may be used to store and access personal files. As we begin to treat all these devices as a single computing environment it will be useful to maintain some of these organizational methods, such as separating work and personal files. Tags will be especially useful because they provide multiple ways to organize data. For example, all video files can be tagged as “media” while tagging personal videos as “family”. These
Tags can then be used for caching and synchronization and also for associating data with workspaces.

**Versioning** would provide a history of each of the users files. *Versioned* files would maintain a history of the changes made so that any prior version could be retrieved. Alternately, a *Changelog* could be used and would only record metadata such as the date, time, and checksum of each modification. Changelogs could be used to aid in synchronization when storage backends are disconnected and reconnected. A combination of these methods may also be used where the number of copies or the length of the Changelog is limited in order to save space. Additionally, temporary files may not need versioning at all.

The sample implementation of the file manager only supports searching and does not yet implement tagging or versioning. The search feature is currently used by the text editor application, as shown in Figure 6.5. When the user wants to edit an existing file, a dialog is shown containing a search box and a list of matching files. When a search term is entered into the search bar the text editor sends the query to the file manager and the file manager responds with the matching entries which are then shown in the text editor's open dialog. The search results contain the name of the storage driver and the file path for the matching
files. When the user selects a file, the text editor then loads the file directly from the storage driver.
CHAPTER 7

Conclusion

In this thesis, we introduced a concept for treating many different devices as a single computing system. We explored a wide variety of specialized embedded and mobile devices that are becoming available and anticipated how these devices will be used in the future. We then developed a one-app model that allows writing software that can make use of multiple devices and can adapt as new devices are discovered and existing devices are disconnected. A message-based communication protocol was proposed to allow communication between these devices and a secure boot mechanism was introduced which allows us to trust the devices being used. We evaluated these methods by proposing several sample applications and implementing them using a one-app model and analyzing what new features and abilities they can provide in a many-device environment.

These proposed methods represent a solution to a problem that may be inevitable. Computing technology will continue to advance and the number and variety of devices will expand to encompass all parts of our lives. As this happens, the existing practice of developing separate applications for each new device will become increasingly complex and unmaintainable. By looking to the future we can begin to lay the groundwork for new programming paradigms that allow interaction and flexibility among devices in order to create a more valuable computing environment.
REFERENCES


