Title
Adaptive Graphical User Interfaces for Custom-Tailored Applications

Permalink
https://escholarship.org/uc/item/8121d250

Author
Pareja-Lecaros, Alberto Octavio

Publication Date
2016

License
CC BY-SA 4.0

Peer reviewed|Thesis/dissertation
Adaptive Graphical User Interfaces for Custom-Tailored Applications

THESIS

submitted in partial satisfaction of the requirements
for the degree of

MASTER OF SCIENCE

in Information and Computer Science

by

Alberto O. Pareja-Lecaros

Thesis Committee:
Professor Richard N. Taylor, Chair
Professor Andre van der Hoek
Professor James A. Jones

2016
# TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>LIST OF FIGURES</td>
<td>iii</td>
</tr>
<tr>
<td>ABSTRACT OF THE THESIS</td>
<td>iv</td>
</tr>
<tr>
<td>CHAPTER 1: Introduction</td>
<td>1</td>
</tr>
<tr>
<td>CHAPTER 2: Background</td>
<td>5</td>
</tr>
<tr>
<td>Adaptive User Interfaces</td>
<td>5</td>
</tr>
<tr>
<td>Mobile Code</td>
<td>17</td>
</tr>
<tr>
<td>COAST</td>
<td>19</td>
</tr>
<tr>
<td>MOTILE</td>
<td>22</td>
</tr>
<tr>
<td>CHAPTER 3: Technique</td>
<td>24</td>
</tr>
<tr>
<td>Building an Adaptive GUI</td>
<td>24</td>
</tr>
<tr>
<td>Practical Use Cases</td>
<td>29</td>
</tr>
<tr>
<td>Justification</td>
<td>34</td>
</tr>
<tr>
<td>Limitations</td>
<td>36</td>
</tr>
<tr>
<td>CHAPTER 4: Demonstration</td>
<td>39</td>
</tr>
<tr>
<td>COASTCalc Web Application</td>
<td>39</td>
</tr>
<tr>
<td>COASTCalc Actor</td>
<td>46</td>
</tr>
<tr>
<td>Client Implementations</td>
<td>53</td>
</tr>
<tr>
<td>CHAPTER 5: Analysis</td>
<td>59</td>
</tr>
<tr>
<td>Client Adaptations</td>
<td>59</td>
</tr>
<tr>
<td>Application Adaptability</td>
<td>61</td>
</tr>
<tr>
<td>Client Ecosystems</td>
<td>64</td>
</tr>
<tr>
<td>Using COAST for Adaptive GUIs</td>
<td>65</td>
</tr>
<tr>
<td>CHAPTER 6: Future Work</td>
<td>71</td>
</tr>
<tr>
<td>CHAPTER 7: Conclusion</td>
<td>74</td>
</tr>
<tr>
<td>BIBLIOGRAPHY</td>
<td>76</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 2.1</td>
<td>COAST Example</td>
<td>21</td>
</tr>
<tr>
<td>Figure 3.1</td>
<td>Message Interpretation</td>
<td>26</td>
</tr>
<tr>
<td>Figure 3.2</td>
<td>REMOTE Message Example</td>
<td>28</td>
</tr>
<tr>
<td>Figure 3.3</td>
<td>Abstract Overall Message Architecture</td>
<td>29</td>
</tr>
<tr>
<td>Figure 3.4</td>
<td>Message Architecture for Web Applications</td>
<td>32</td>
</tr>
<tr>
<td>Figure 3.5</td>
<td>Multiple Actors Serving Multiple Web Applications</td>
<td>33</td>
</tr>
<tr>
<td>Figure 3.6</td>
<td>Unserializable Lambda Example</td>
<td>38</td>
</tr>
<tr>
<td>Figure 4.1</td>
<td>Basic COASTCalc Web Application</td>
<td>40</td>
</tr>
<tr>
<td>Figure 4.2</td>
<td>Dijit Initialization Code</td>
<td>41</td>
</tr>
<tr>
<td>Figure 4.3</td>
<td>AddDigit Function</td>
<td>42</td>
</tr>
<tr>
<td>Figure 4.4</td>
<td>Operate Function</td>
<td>44</td>
</tr>
<tr>
<td>Figure 4.5</td>
<td>AddOperation Function</td>
<td>45</td>
</tr>
<tr>
<td>Figure 4.6</td>
<td>Basic Calculator HTML Markup</td>
<td>46</td>
</tr>
<tr>
<td>Figure 4.7</td>
<td>COASTCalc Actor Interaction Architecture Diagram</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4.8</td>
<td>COASTCalc CURL Definition</td>
<td>47</td>
</tr>
<tr>
<td>Figure 4.9</td>
<td>Message Receiver Activity Diagram</td>
<td>49</td>
</tr>
<tr>
<td>Figure 4.10</td>
<td>Web Server Thread Spawn</td>
<td>50</td>
</tr>
<tr>
<td>Figure 4.11</td>
<td>Appending Whalesong Generated Files Code Snippet</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4.12</td>
<td>Whalesong to Existing JavaScript Bridging Code</td>
<td>51</td>
</tr>
<tr>
<td>Figure 4.13</td>
<td>Injecting CSS modifications to the existing HTML Markup</td>
<td>53</td>
</tr>
<tr>
<td>Figure 4.14</td>
<td>COASTCalc Actor Interaction Architecture Diagram</td>
<td>54</td>
</tr>
</tbody>
</table>
Figure 4.15  ‘basic Message Definition 55
Figure 4.16  Modulus Definition In Racket 56
Figure 4.17  ‘addFunction Message Definition 56
Figure 4.18  COASTCalc Web Application With Modulus Function 57
Figure 4.19  CSS Modification Definition 57
Figure 4.20  ‘addCSS Message Definition 58
Figure 4.21  COASTCalc Web Application With CSS Modification 58
Building graphical user interfaces has been a continuing challenge for software developers and designers ever since graphics were introduced in computing. These days, in the world of ubiquitous and mobile computing, building user interfaces to work on every device and platform is tedious at best and requires duplicating work. To tackle this challenge, we introduce an approach which allows clients to modify a software application's user interface without the application knowing about any of those clients beforehand. Our technique employs the use of mobile code between clients and the application in order for those clients to adapt the application’s user interface to best fit their needs. We use the ComputAtional State Transfer architectural style in order to simplify the design of the communication layer between clients and the application and to ensure secure communication between the entities.

We find that building such an application can be practical and that clients are able to introduce code to the application such that the graphical user interface is adapted to those clients. To demonstrate, we created a calculator application whose graphical user interface and related functionality is modified and then consumed by three different clients. The first
client uses the application as is, the second client introduces additional functionality to the application, and the third client introduces a change to the presentation of user interface components in the application in order for it to work better on its devices. All three clients are accommodated without the application having any knowledge about what modifications the clients specifically make.
CHAPTER 1

INTRODUCTION

As computing devices become more ubiquitous in society, it's becoming harder and harder to create applications and services that can properly interface with the wide range of software and hardware combinations that these devices exhibit. Devices of varying form factors, screen sizes, screen resolutions, and graphical user interfaces (GUIs) are now expected to be able to access the same software applications [1]. What's worse, the future looks to bring traditional appliances and accessories into the ecosystem of computing devices. Furthermore, these devices are expected to coexist and share services between each other's software systems. So, as computing technology continues to evolve, how best to create services and applications to keep up with the ever-expanding ecosystem of possible devices?

Currently, in order to provide a service that can be used across multiple device types, organizations have to employ teams of software developers that are each specialized to handle one particular type of device or platform, whether it is a smartphone running Android, an iPad, or a desktop computer running Linux. These ecosystems each present their own unique challenges. Android devices are not uniform in size nor form factor, requiring that software designers keep these differences in mind while writing code that can be run effectively on all of those devices. Any GUIs and layouts must be designed to be flexible enough to handle inconsistent screen sizes, screen resolutions, and different form factors (physical keyboard versus digital keyboard, for example). Even iOS devices have differences across the different versions of a product line, such as the iPhone. The iPhone 5
introduced a new ppi (pixels-per-inch) standard for their displays than previous versions [2], requiring that software designers adapt their GUIs to make sure it works well with both the old and new ppi specification.

It doesn’t get much better in the world of desktop computers. A typical desktop computer can never guarantee any screen size since most operating systems use a window manager that allows running applications contained within a window to be resized. For example, for web applications that run in a web browser, responsive CSS is a solution towards creating a GUI that is dynamic enough to work with various discrete screen size ranges as specified by the designers of the GUI. Often it isn’t enough to simply tweak the attributes of UI elements in the CSS and additional HTML elements are often added to a page and then are hidden or shown depending on what CSS is being applied for a particular screen size. This is because sometimes it’s more difficult to manipulate a particular GUI element’s attributes to work with all screen sizes, or it is impractical to change the layout of UI elements simply by changing the attributes of elements rather than changing the HTML markup itself. By combining all of the different screen size UI elements and layouts in the same HTML markup, the GUI is now tightly coupled between each screen size, making it potentially difficult to make a modification for one screen size without it affecting the other screen sizes.

This thesis presents a technique for tailoring the functionality and corresponding GUI components of applications served from software services to meet different clients’ needs. Previous work suggests that a service consumer can program service providers in order to produce exact manifestations of services they wish to provide [3]. We have coopted this idea to allow clients to install new functionality into an existing service,
allowing that service to produce a custom-tailored software application. In this scenario, the server acts as a service provider which hosts an application and any custom-tailored versions of that application. The client acts as a service consumer, introducing functionality to the server to be adapted into an existing application and its GUI and then consuming that new application. We accomplish this by using COmputAtional State Transfer (COAST), an architectural style that allows us to send mobile code between a client and a service to modify the capabilities of that service in a secure fashion. These capabilities can consist of new functionality or modifications to existing functionality, including changes that ultimately alter any GUIs belonging to a service. We present this technique as a way to allow service designers to control the adaptations that are accepted from clients while allowing each client to ultimately create the application GUI adaptations it needs, freeing the service designer from having to consider all contingencies and the individual GUI needs of each client.

COAST provides a mechanism for passing messages that can contain data and closures that define functionality and suggested GUI components. The server is responsible for interpreting those messages and creating the GUI components that will be used to interact with that newly introduced functionality, either by interpreting the suggested GUI component from the message itself or by deciding itself what GUI component should be created. We are working under the assumption that a client best knows its own configuration and thus is the best suited to define a GUI that fits its needs. However, the client need not be responsible for the direct creation of the GUI.

In order to demonstrate this technique, we have built a service called COASTCalc, COASTCalc serves to demonstrate GUI dynamicity by having clients adapt COASTCalc in
various ways. We demonstrate the power clients have in creating a GUI experience that can best be consumed for their needs, whether those needs be due to form factor or the type of device. We’ll show how COASTCalc can receive messages containing environment-agnostic data and then use that as a model for how other services can use the technique for the dynamic creation of new, custom-tailored GUIs.

To begin, we discuss prior work in adaptive interfaces and how our contribution fits within that ecosystem of work. We will introduce the concept of mobile code, its technical properties, and how it will be used in our technique. Following that we delve into a discussion of the COAST architectural style, how it works, and how we will ultimately incorporate its mobile code concepts for our technique to create an adaptive GUI for an application. After our description of the technique, we present a working demonstration, an analysis of our findings during its implementation, and some suggestions for future work.
CHAPTER 2

BACKGROUND

In this section, we discuss the various concepts and technologies that form the basis for our technique. We begin by discussing the work that has been done involving adaptive user interfaces, exploring the various techniques and contributions that have helped advance GUI generation to what it is today. We’ll then dive into the technology we use in order to implement our technique, including a discussion on the underlying architecture that we use to produce a working demonstration.

2.1 Adaptive User Interfaces

Much work has been done adapting user interfaces to meet different technological and user needs. As user interfaces have evolved through the creation of toolkits, layout managers, and frameworks, so has the potential of creating adaptive user interfaces in a wide variety of application domains. In this section we’ll discuss much of the prior research, highlighting the different tools and techniques that have come about this work through different eras of graphical user interfaces, and how these techniques have shaped where we are today and how they have influenced our own technique.

The definition of an adaptive interface is dependent on the context with which the term is used. For our purposes a GUI that can be automatically and programmatically generated is not in and of itself adaptive unless that interface can also be made to handle different user needs, technological needs, or other needs not inherent to the business logic
itself. This differs from a dynamic interface, which for our purposes refers to a GUI’s ability to be manipulated and altered at runtime by either its own code or by the user.

Before the advent of the modern toolkits and libraries for user interfaces built in the plethora of programming languages that exist today, GUI programming was considered a difficult and time-consuming task. In many cases it was difficult to separate business logic with presentation logic, and that presentation logic code could have been as high as 60% [4] of the entire code base. Many systems and techniques arose to challenge this difficult problem, starting with the ability to map the user interface presentation to the underlying data model of an application.

2.1.1 Automatic GUI Display Generation

Creating an adaptive user interface first requires that a GUI can be created or generated automatically. It is not enough for a developer to programmatically create the interface; GUI components need to be automatically created to represent its underlying data in a way that can be interpreted by a user in a meaningful way. This ability to create GUIs in an automated manner has allowed developers to concentrate more of their efforts towards making the user experience more effective for different devices and users instead of worrying about making sure data can be presented at all and in an accurate manner.

It is important to distinguish the difference between an adaptive interface, a dynamic interface, and an automated interface – that is, an interface that has been automatically created. An automated interface allows a GUI to be created in a manner that can represent its underlying business logic and any rules that the business logic might follow. This allows programmers the freedom to design a generic GUI to fit within the
business model and its rules and allow that GUI to work when the rules change the model in predictable ways. An example of this would be to allow a GUI to create a view consisting of a list of items, regardless of how many items might actually exist in the underlying data.

Different techniques have been developed to handle different ways in which a GUI might be automatically generated. Using artificial intelligence rules it is possible to map a business model to its presentation following those given rules, allowing a developer a quicker means of developing an easily modifiable user interface [4]. Scope is one such implementation of a rules-based system designed to use rules that maps a C++ application’s data and functions to the graphical interface [5]. Other approaches separate the look and feel specification of a UI from the specification of what each UI element contains by separating how each entity is defined. Jade is such a tool, allowing a UI designer to create the rules and objects defining the UI to be stored in a look and feel database. Jade combines the application developers’ specifications with the UI rules found in the database to create meaningful dialogs [6].

The issue with many of these techniques is that it still requires a great deal of upfront work in order to specify the UI rules, objects, and interactions, not to mention the boilerplate require to bridge the UI specification with the business implementation. The layer of abstraction provided by creating UIs automatically certainly reduces the amount of UI code that an application developer needs to write, but better would be to have an abstraction or set of abstractions that can be reused among many different applications across application domains and datasets.
2.1.2 Toolkits

Being able to create GUI components automatically is not sufficient for creating sophisticated GUIs. Maintaining the relationship between the underlying data structures of an application and its UI elements, especially once user interaction is involved, requires a lot of boilerplate code. By placing constraints to support these relationships [7], it becomes possible to define a set of abstractions that can be used by many different application GUIs to tie specific business logic to graphical elements. Instead of having application developers reinvent those sets of abstractions and their graphical elements, toolkits provide a library of widgets such as buttons and scroll bars that define all of the abstractions necessary to allow the interaction between those graphical elements and application-specific business logic.

Coral is one of the earlier toolkits to use graphical objects and constraints to support the construction of advanced user interaction techniques such as menus and scrollbars [8]. Hierarchical graphical objects are defined using object inheritance to define more specific objects that are based on the parent object. This allows for flexibility when using the toolkit so that UI objects can be better fitted to meet user interaction needs.

Nowadays a toolkit or framework takes care of many of the user interface building components and it is easier than ever to separate business logic from presentation. Instead of manually creating each graphical component by hand, the framework comes with a set of pre-existing UI elements that can be used in an abstract manner for any application written in that toolkit or that framework's programming language. Examples of such toolkits include the Java Swing library [9], and Cocoa Touch [10].
A common downside of toolkits is that they are usually language-dependent, even the ones that are platform independent. For example, Java is a platform-independent language, meaning that its bytecode will run on many common operating systems. Java’s Swing library makes the same guarantee, allowing it to work on all of those platforms with minimal differences in the way the UI elements are displayed. However, there is no way to use the Swing library with a language that cannot interpret Java bytecode, and so the toolkit is otherwise unavailable to application developers working in other programming languages.

Toolkits certainly reduce the time it takes to build graphical components for applications, but they do not address a UIs dynamicity or ability to be adapted. Still, it is up to the application developer to build those features. Alternatively, a UI framework or library might contain components that can handle certain levels of dynamicity.

2.1.3 Layouts

Layouts are used to methodologically control the placement of UI components on a screen in an abstract fashion, ridding GUI developers from the tedious task of figuring out how to display UI components based on different screen sizes, resolution, and coordinate positions on a screen. Layout managers define different layout rules and constraints a developer has access to and hides the implementation of those rules and constraints from the developer. This frees the developer from worrying about specific system or language-specific details for the positioning of GUI elements and lets the developer work on creating an interface that best meets the needs of the consumers of that interface.

The GADGET toolkit is a set of three abstractions that attempts to optimize the generation of a display by iterating and evaluating different generated solutions for UI
interfaces [11]. This set of abstractions work together as a layout manager, combining algorithms with its optimization approaches to produce layouts that fit certain testable criteria, such as minimizing the amount of empty space used or making sure that labels describing UI elements are kept together.

Java's Swing library contains not just a toolkit for UI components but also a layout manager that can produce a wide variety of layouts. Each layout comes with its own rules and constraints for how they will display different GUI elements and are abstract enough to work with any of Swing's UI components. Most modern layout managers are similarly built but for their respective programming languages. They make the otherwise tedious task of programming the positioning of GUI elements a lot easier and reduce the complexity of UI code from the overall application code base.

2.1.4 Multi-Device Interfaces

All of the constraints, rules, and assumptions that toolkits and layout managers consist of do not often consider the wide spectrum of computing devices that a GUI might need to support. Even if a layout were to consider certain technical constraints such as screen size, there's no way for the layout to inherently alter its internal layout rules accurately for different screen sizes across all GUIs and business requirements. Different GUIs end up being created for different devices, increasing the amount of work required to have an application run on all of the platforms it would like to support.

Many architectures and frameworks have been developed in order to adapt a GUI to different devices. One approach focuses on creating rules that govern how a layout might adapt to different screen sizes. An example of such a technique uses annotations created by
a UI designer in order to influence algorithms within an adaptation engine to create layouts that best fit hardware requirements [12]. That same approach is used by UiBuilder, which uses XML as its description language for annotating UI elements [13]. Another approach uses an architecture that looks to transform the graphical interface into another interface. It allows the graphical portion of the interface to be abandoned entirely, mapping the UI components to an auditory interface instead. Such a technique can be useful for certain hand-held devices with screen real-estate that may be too small for the application to effectively use, hand-held devices with no graphical interface whatsoever, and for users with visual impairments [14].

Transformation techniques require that constraints be assumed. One possible combination of constraints that could generate an acceptable GUI for small screens might look as follows:

1. Maximum use of the available space
2. Minimum amount of navigation clicks
3. Minimum scrolling (except list widgets) [15]

The above constraints might be acceptable for certain classes of smaller screen devices, assuming that those devices have a similar form factor to a laptop or desktop – meaning a keyboard and a mouse. Clicks do not mean much to a touch screen, and it is possible that GUI elements are created too small to press. Furthermore, it may be counterintuitive to limit the amount of scrolling on a touchscreen if doing so means that all of the GUI elements are crammed into the viewable window. Even if additional constraints
were added that placed limits on how close together UI elements can be positioned, or constraints that placed limits on the minimum size of different components, there is no guarantee that those constraints will work for even the majority of small screen devices. Additional assumptions have to be made about the minimum distance between UI elements as well as assumptions about the minimum size of those elements in order for such constraints to be effective. It is also possible that an entirely different set of UI elements better fit an application's interface needs.

2.1.5 Mobile Interfaces

Internet-enabled mobile phones allow users to access information and browse websites from nearly anywhere. The problem with such devices is that due to their form factor and relatively small screen, websites optimized for desktop computers provide poor experiences for mobile users. The usability of the browser model was inherent to the desktop’s capabilities, largely the size of the screen, a mouse and keyboard, as well as relatively high bandwidth and CPU resources [16]. The solution then is to build websites whose design accommodates mobile users.

Many websites have both a desktop version and a mobile version. These can be easily recognized by the difference in the URL. The website will direct a user to the appropriate version of the website that it thinks will best suit that user by reading the User-Agent string the client provides. That string is then parsed and a determination is made on the type of device the client is using. Sometimes the website provides the option to view the desktop version anyway when on a mobile device, but that choice is left up to
the user. In general, the two UIs are created separately, doubling the amount of work required to create a reasonable user experience for both desktop users and mobile users.

Another approach for having websites work on both mobile and desktop computers is to build tools specific to each of those devices to provide an optimal experience. Power Browser was one such tool that creating a browsing experience suitable for PDAs [17].

While the work is focused on viewing and manipulating information located on the World Wide Web, the same concept can be applied to any service hosted anywhere. Today, mobile versions of browsers exist with features that enhance navigation of web pages, including zooming in on text links before one is selected in case of an ambiguity in input detection (a person’s finger hits multiple links at the same time) and allowing touch gestures to manipulate text in text fields.

Clearly mobile browsers have not fully solved the problem at hand. While tools to ease the mobile experience helps with some of the experience of interacting with a website, they do not solve any inherent issues with the GUI that a website displays. The specifics for how to optimize each individual website cannot be satisfied by the browser alone and must be handled by each individual website or service. Having the browser be the one responsible for the mobile UI experience would be akin to changing the HTML specification to improve the layout of websites – it simply cannot be done in a manner that would benefit the entire ecosystem.

The Ubiquitous Interactor (UBI) is another attempt at creating custom-tailored services for different devices. It works by separating the GUI from the service by abstracting user-service interactions and presentations into their own entities [18]. UBI consists of three parts: The Interaction Specification Language (ISL), customization forms,
and interaction engines. ISL is used to encode interaction acts, which serve as the communication between devices and services and express user-service interactions. Interaction acts serve as device-independent communication between devices and services. Interaction engines are device-dependent and are what allow a user to use a particular service on a device. Finally, customization forms are both device-dependent and service-dependent and allow a service some degree of control of the presentation of its services for a particular kind of device.

UBI defines a level of abstraction that allows user interactions with a service to be tailored towards different devices; however, the user interface remains dependent on any particular combination of a service and a device. This does not differ greatly from reading the user agent string in a browser to detect the type of device a client is using to access the service. The technique requires that device-GUI pairings are created in advance, duplicating the amount of work required to render GUIs appropriate to different devices types.

Phones are not the only mobile devices that have access to the Internet. Many different devices that are Internet-enabled are now ubiquitous, providing a wide-range of capabilities and experiences, as well as UX challenges to overcome. Additionally, these devices need not necessarily operate standalone. For instance, the Apple Watch communicates with the iPhone to bring an iPhone’s notifications to the watch [19]. Currently there is no means by which a browser might be able to support a GUI that will work for the Apple Watch no matter how small the UI elements get or what layout is chosen simply due to form factor and screen window size limitations. As such, native
mobile applications have been the prominent means with which to combat the inadequacies of browser-based web technologies on these devices [20].

2.1.6 Responsive Design

Native mobile applications have not negated the need to have web experiences that are optimized for different devices, especially due to the advent of mobile browsers. Building native mobile applications also takes a different skillset than building a website with traditional technologies such as HTML and JavaScript, meaning that not only is effort duplicated maintaining two different applications for the same service, it may require additional personnel with the necessary skills to build the native application. The effort is again repeated for each native mobile platform being targeted. It would be convenient to have the ability to take the platform-independent web and make it serve all of those different devices at once, avoiding the need to learn new platforms, languages, and duplicating effort.

Responsive design allows the presentation of a webpage to be altered based on the size of the window detected by the browser. Responsive CSS (Cascading Style Sheets) enhances the CSS syntax with HTML5 media queries, which define different CSS styling based on discrete window sizes [21]. This allows a single CSS file to define the presentation of both a window made for a traditional desktop screen as well as for the screen of a mobile device. Alongside HTML attributes, this can lead to an optimized presentation for a website within the defined screen sizes. It has become a pervasive technology that is used today to produce web content that can be viewed and interacted with on multiple devices [22].
There are problems to the responsive approach. First, it tightly couples each presentation definition, making it potentially very difficult to make a change for one screen size without it affecting another. To illustrate, imagine a CSS rule that is common to multiple screen sizes. If that CSS rule needs to be modified for one screen size, then the other screen sizes will be impacted as well. All of a sudden, every single screen size configuration has to be taken into account in order to make the required change. Second, there is no guarantee that an HTML’s layout can adequately handle every screen size. It is possible that different structures are necessary to accommodate different screen sizes. One could theoretically introduce every structure necessary for each screen size by hiding elements once they no longer apply to a certain layout, but then you run into the same coupling issue that the CSS has, but worse. The CSS file at least makes it clear which CSS rules apply to which window sizes. HTML markup has no notion of window size and it can be jarring to see a bunch of elements that are not likely to appear right away. Third, the above problems assume that the implementer of the GUI is also the designer. A UI designer may not take into account different window sizes, affecting the implementation of the GUI code when trying to make sure the interface is usable for all of the window sizes it supports. A mockup tool that can automatically convert a designed mockup into usable code might alleviate some of the inherent issues that a designer and a developer might have when putting together a GUI [23]. However, generated code in this manner is often very difficult to read, making it difficult to change later.

There are other ways to use CSS3 media queries to allow customization of GUIs without relying on the HTML5 specification alone. Letting clients adapt a layout to best suit their preferences allows GUI designers to build their layouts in a more extensible way by
not having to worry about potentially infinite combinations of client specifications.
CrowdAdapt looks to solve this issue for web applications by allowing each individual user
to define their own layout preferences and for those preferences to be saved such that
others may use those preferences or further adapt them [24]. Their adaptation engine uses
CSS3 media queries under the hood to define the styling as the user manipulates the layout.

The idea is interesting, but leaving it in the hands of the end-user might not be the
most appropriate approach in all circumstances, and potentially under no circumstances.
The end-user is not always the right choice for making design decisions. Furthermore, it
does not address functionality that may need to be altered to create a better experience on
a particular device. Still, it shows potential in allowing the clients themselves to create
their own GUI experiences that could never have been anticipated as optimal from a
developer or designer’s standpoint.

2.2 Mobile Code

Mobile code addresses service customization and dynamic extension of application
functionality [25]. Service customization allows for client-specific tailoring of functionality
and its corresponding user interface. As an example, mapping software built for a desktop
may have a corresponding version for a device with GPS capabilities, such as a mobile
device. The mobile device can send computations that introduce its GPS functionality to the
service, allowing for an initial rendering of the map to center on the exact location of the
device. Furthermore, it can adapt the map GUI to render an icon which keeps track of its
location.
Mobile code can be categorized as both strong and weak. With strong mobility, code, data, and execution state are all transferred from a host machine to a target machine. That is, one machine could run some computation, pause it, and then transfer the code and any data involved with that computation to another machine which continues to run the computation. With weak mobility, only code and data are transmitted [25]. Our technique highlights weak code mobility but is not necessarily limited to it.

**Moving Code** – The ability to move code allows for messages containing new capabilities and functionality to be installed and executed on the target machine. For example, messages can include definitions of GUI components and their functions that can be installed on the target machine for later execution. Mobile code can introduce new GUI toolkits or individual, standalone GUI components. The target machine can then either integrate the code received as-is if it’s already written in the target programming language or use a compiler to translate it into useful instructions. It is up to the system developers to determine what sorts of mobile code should be allowed to be executed.

**Moving Data** – The ability to move data allows a target machine to interpret that data in a way that the data originator could not. For example, receiving data about a client’s screen size could allow a service to adjust its application’s GUI components such that they are better usable by that client. Receiving data about a device’s form factor can also inform the GUI as to possible improvements to its interface that are best tailored for that device. Just as with moving code, it is up to the system’s developers to determine what data the system will accept and how it will interpret that data. Often new code that has been installed will
allow the system to accept and interpret new kinds of data, and that possibility has to be carefully considered by the system’s developers as well.

Clients and servers will use mobile code to communicate back and forth in order to allow clients to adapt a service’s applications and their respective GUIs in order to better consume that service. By allowing this level of communication, a software developer no longer has to anticipate the exact needs of each individual client and can instead focus on building a sufficiently abstract service that allows a GUI to be easily adapted by interested clients. However, there are problems associated with mobile code, most notably in the areas of security and trust. A service that provides a dynamic GUI that is vulnerable to malicious clients is intolerable, so the system must be designed such that it can be used by trustworthy service consumers or that can limit potential attack vectors to less trustworthy service consumers. Thankfully, there exists software architecture capable of managing security and trust in a non-obtrusive manner, freeing the software developer of such issues. That architecture is called COAST.

2.3 COAST

COAST (Computational State Transfer) is an architecture that derives its origins from CREST (Computational REST). CREST takes advantage of mobile code and continuation passing style in order to exchange more advanced computations than one might see using AJAX [26]. These computations are called actors and consist of a closure, a continuation, or a binding environment, which can be defined as a key-value pair mapping [3]. COAST is the
next evolution of the concept of computation-exchange, and can be described by the following axioms:

- “All services are computations whose sole means of interaction is the asynchronous messaging of closures (functions plus their lexical-scope bindings), continuations (snapshots of execution state) and binding environments (maps of name/value pairs [27])
- “All computations execute within the confines of some execution site \(<E, B>\) where \(E\) is an execution engine and \(B\) a binding environment
- “All computations are named by Capability URLs (CURLs), an unforgeable, tamper-proof cryptographic structure that conveys the authority to communicate
- “Computation \(x\) may deliver a message (closure, continuation, or binding environment) to computation \(y\) if and only if \(x\) holds a CURL \(u_{y}\) of \(y\)
- “The interpretation of a message delivered to computation \(y\) via CURL \(u_{y}\) is \(u_{y}\) – dependent” [3]

COAST messages operate on two basic mechanisms, REMOTE and SPAWN. A REMOTE command allows a computation to be passed to a target machine for evaluation. The result of the computation is then returned to the sender. With a SPAWN command, a message is sent to a target execution host and a computation incorporates the code and data from that message as a service on that host. The execution host returns a CURL naming the location of the newly installed service, called the execution site [26].

To illustrate the use of COAST messages, Figure 2.1 diagrams a payment provider interested in integrating its payment method with an online store begins by obtaining a CURL allowing it to communicate with that online store. This issued CURL is given to the payment provider in some pre-negotiated manner, such as through a business contract. Using the store’s CURL, the payment provider can issue a SPAWN message containing functions defining its payment solution service. The store interprets the SPAWN message...
and installs those functions on a new actor, creating a CURL that allows other interested parties to communicate with the newly installed payment service. Later, when a client wants to try to make a payment after shopping at the store, the store can now pass the CURL with the location of the payment service, allowing that client the ability to make payments using that particular payment provider.

Figure 2.1 – Introducing a new payment provider to an existing store. Clients will ultimately be given access to the provider through the store to make payments.

Another important aspect of the COAST architectural style is that messages are interpreted with the confines of an actor’s binding environment. An actor’s binding environment serves as a mapping of all possible capabilities of that actor’s execution site. A client that sends a message to a server for example will have that message be interpreted
by the capabilities and restrictions of the server's binding environment and not by how the client would like its message interpreted. This is important in allowing servers the ability to prevent potentially dangerous modifications to server resources but still allow clients to communicate with a service by sending messages that tailor that service to fit the clients' needs.

The ability to communicate via the exchange of messages between typically non-associated actors is not without its inherent security risks brought about directly from the use of mobile code [25]. Such risks include the mishandling of system resources (causing denial of service or an unnecessary waste of resources), the execution of code that introduces a bug into the system or unveils a bug that puts the whole system into an undesirable execution state, or a trusted source could inadvertently misuse a service, again putting the service into an undesirable execution state. As such, it is vital that security be at the forefront of architectural concerns [28], something that COAST promises through its Capability URLs and the Principle of Least Authority on which these CURLs work.

2.4 MOTILE

MOTILE [3] is a functional mobile code language implemented in the Racket language that serves as the implementation for COAST CURLs, computations, and messages. All MOTILE actors occupy homogeneous address spaces called islands, which act as execution host implementations of COAST. All MOTILE data structures are immutable because they are functional, thus simplifying the distinction between inter-island communication and intra-island communication of MOTILE actors. Furthermore, this
property provides an inherent security feature, preventing attacks that manipulate the values of data after it has been shared between actors. All CURLs are issued by some island and its properties are enforced by the issuing island. Basic enforceable criteria includes the total number of times a message can be sent using a particular CURL, at which point in time a CURL is no longer valid, the rate at which messages can be transmitted per CURL, and whether or not a CURL has been revoked, denying any actor the ability to use that CURL for communication.

For our technique, the COAST architectural style will be the underlying architecture for communication between clients interested in making GUI adaptations to an application and the service that provides that application. Specifically, we use MOTILE and its ISLAND infrastructure to create client islands that can define their own functionality and GUI modifications in a meaningful way – that is, in a way that a service island can interpret and install. By leveraging the mobile code capabilities of MOTILE, we can apply service customization and dynamic extension of application functionality at the user interface level. We do this by using MOTILE’s CURL implementation, allowing us to send messages that encapsulate computations and any metadata that can be used to define new GUI components and their workings to the target service.
CHAPTER 3

TECHNIQUE

In this chapter we present our technique for constructing adaptive graphical user interfaces. First we begin by explaining the technique itself and how it results in the creation of an adaptive graphical user interface. We'll discuss how we developed the technique, including the architecture and software design decisions that were made. Specifically, we explain the use of MOTILE and the COAST architectural style as the framework for enabling GUI adaptations defined by clients onto an existing application, ultimately tailoring the application to fit their needs. Furthermore, we'll show how this technique can be applied to different kinds of applications. Finally, we justify this technique against existing techniques and present limitations with our approach.

3.1 Building an Adaptive GUI

Our work entails the adaptation of an application GUI by allowing it to be modified by clients in order to custom-tailor the application’s GUI for each client. Adaptability is defined as “a software system's ability to satisfy new requirements and adjust to new operating conditions during its lifetime” [29]. As such, we allow clients to define new requirements for an application’s GUI instead of having the designers of the application make the adaptations, provided that those clients are 1) given the ability to communicate with an actor that can make modifications to the GUI, and 2) that those clients are given the means with which to communicate with the actor in an effective manner (e.g. an API of message formats, documentation of how messages will be interpreted by the actor, and so
A GUI that can accept a change in business logic not present when the GUI is first generated is considered adaptive.

What might adaptations created by clients consist of? An adaptation generated by a client for our purposes is restricted to 1) the introduction of a new GUI component to an application, 2) the modification of an existing GUI component in an application such as its style attributes, or 3) the modification of the layout of GUI components, including the replacement of one layout for another. These adaptations can be applied to an individual GUI component or an entire class of GUI components at once. Such classes of components include buttons, checkboxes, or layouts defined by a layout manager. A client’s need to make such modifications typically comes from the needs for its end users. A client may need to make GUI components larger or smaller to fit a custom screen size, or a client might need to add an additional GUI component to make up for an input control that an end-user’s device might not have. Although we’ve restricted the adaptability of the GUI to a specific set of criteria, an application developer can choose how adaptive an application’s GUI should be by specifying the scope of the adaptations allowed.

In order for client-defined adaptations to make meaningful changes to an application’s GUI, some execution thread must contain logic that can interpret messages from clients that correspond to the adaptations a client would like implemented. The number of ways messages can be interpreted and how those messages are interpreted are what define the scope of adaptations allowed and are thus what determine the dynamicity of the GUI. In Figure 3.1, a thread that receives client messages has been designed to accept three different kinds of adaptations defined through a message identifier: addComponent, changeLayout, and changeSize. Each identifier is handled by executing different functions.
defined locally by the service, scoping the data and mobile code found in the received messages to code that has been locally defined.

```java
switch message.id {
    case "addComponent":
        addNewGuiComponent(message.component)
        break
    case "changeLayout":
        changeLayout(message.layout, message.data.layoutParams)
        break
    case "changeSize":
        changeComponentsSize(message.component, message.data.width, message.data.height)
        break
}
```

Figure 3.1 – Pseudo-code demonstrating the possible interpretations of a client message based on the message’s id. Any received messages not matching any of the ids are simply discarded. Furthermore, local functions (addNewGuiComponent, changeLayout, and changeComponentSize) provide the logic for how a message’s data is read and then interpreted as modifications for the GUI.

If a client would like to change a GUI in a way that does not add a component, change a layout, or change the size of an existing component, then the service, and ultimately, the GUI, is not adaptive enough to accommodate that adaptation. Additionally, it is up to the addNewGuiComponent, changeLayout, and changeSize functions to effect changes to the application’s GUI in a way the client intended.

A service component houses the execution thread whose purpose is to 1) receive messages from clients wishing to make adaptations to the application’s GUI, 2) interpret those messages and make changes to the application GUI, and 3) send response messages to those clients with data containing information about how to access the modified application and its GUI. We take advantage of MOTILE and its island infrastructure’s implementation of the COAST architectural style to allow clients and the service component to exchange messages safely. A service is implemented as a MOTILE island with at least one actor that acts as the aforementioned execution thread. Each client in turn is also a MOTILE island, each with an actor that can communicate with the service actor if and
only if the client's actor has obtained a CURL that allows it to communicate with the service actor. A client actor is responsible for defining some modification in a lambda that's written in a language understood by its execution context. That actor then sends a lambda, its own CURL, and other metadata in a message to the service. The service actor receives the message, deserializes the payload, interprets the metadata found within the message to determine a course of action to take, and then executes that action as defined within its own execution context.

As an example, consider a client that would like to make presentation changes to a web application. After defining some CSS within its own execution context, it creates a message triple \(<c, m, \lambda>\) where \(c\) is the client actor's CURL that the service actor can use to send a response, \(m\) is metadata containing a message identifier that the service actor understands, and \(\lambda\) is a lambda containing the CSS description. In Figure 3.2 the sender component sends this triple as a REMOTE message to the service actor. The service actor receives the message, interprets the message identifier from the metadata and does whatever internal logic it has to handle such a message, and then sends a response using the provided CURL containing the results of the action performed. Typically, the results contain some physical address of the application or information about how to access the modified application. The payload making up the result can consist of serialized data or native code depending on the type of application and the configuration of the service and client islands as determined by the business context.
Figure 3.2 - A client sends a REMOTE message to a service and gets a response. CSS definitions created by the client are transmitted to the service and then interpreted based on the logic contained on the service actor. A response containing useful information for the client is returned, typically including at least the location of the modified application or GUI.

The application itself is a local resource on the service island that consists of one or more files and execution threads that is directly accessible by a service actor in its execution context. The service actor can make changes to the application using one of the following methods:

1) Directly accessing files on the local file system and altering the files
2) Passing data to the application via native application code
3) Sending data to a listener on a port created by the application

Clients never make changes to the application’s GUI by directly communicating with the application and have no way of directly communicating with the application’s GUI thread. In figure 3.3 we show the flow of data required for applying an adaptation to an application’s GUI without direct communication between clients and the application. Client actors send messages as triples \( \langle c, m, \lambda \rangle \) to actor \( a \) running on the service island. Those messages are interpreted by actor \( a \), and the resulting data is used to modify the application using one of the methods described previously. To do so, actor \( a \) creates a thread Main that acts as the main execution thread for the application. Main then creates two more threads, a GUI thread for displaying the GUI and a Listener with which actor \( a \) can communicate. Actor \( a \) sends an adaptation from the client to the Listener thread in code that the application can execute. The listener thread then executes the code,
modifying the data in the application GUI thread. Actor \( a \) then sends a response to the client actor with information about how to access the modified application. The actual communication method for exchanging data between actor \( a \) and the application is implementation specific.

![Diagram](image)

Figure 3.3 – Clients send MOTILE messages to a service through the island infrastructure. The service actor creates a thread Main that begins the application. The application creates both a GUI thread and a Listener that allows the service actor to communicate changes to the GUI. The Listener thread passes data from the actor to the GUI thread in order to enact any changes.

### 3.2 Practical Use Cases

How can we apply this technique in a practical way? What kinds of applications can this technique be applied to? Where do the CURLs to communicate with these services come from, and how can this technique be used without causing harmful side effects to the application in which the technique is applied? The answers lie within the constraints ensured by the COAST architecture as implemented by MOTILE in tandem with design decisions that make an application’s GUI adaptive enough to be modified by clients in a purposeful way.
We can apply this technique for a desktop application that runs on a single machine. As Figure 3.3 demonstrates, a service island contains an actor that accepts communication from other actors. The service island in this case is also responsible for running the application in a separate execution thread using native code. Because the service island is already running on the local machine, the end user can interact with the application as-is. Once the main application thread is running, it creates a listener that listens for data from the service actor such that GUI changes can be made. The interface definition for communication between the service actor and the application is determined upon design of the system and the method for communication can range from native code to serialized data using a TCP connection through a local system port. A client island is also created on the same machine. The client island can choose to make GUI modifications to the application if necessary, sending instructions that the service can interpret in order to make GUI adaptations. Using MOTILE’s infrastructure, the client actor sends a message to the service actor using the service actor’s CURL, which allows the client to make some GUI modification to the running application. The service actor takes the message, interprets it using its internal logic, and then makes the GUI adaptation by passing data to the application listener. The application listener is responsible for executing the received code, creating changes to any affected GUI components. When the GUI refreshes the changes are applied and the end-user sees a corresponding change to the GUI.

We can also apply our technique for use with basic web applications by making minor modifications to how an application is updated and accessed. Figure 3.4 shows the flow of communication between client actors and the service actor just as with the desktop application. A client actor sends a message of the form \(<c, m, \lambda>\), where \(c\) is the client CURL,
contains a message identifier that the receiver can use to interpret the message, and \( \lambda \) contains a closure to be executed by the receiver or the application. The service actor decides how to interpret the message based on the message identifier found in the payload and then translates the enclosed lambda into code that be inserted or executed by the execution context of the application. Because a web application works by transferring files to a remote process to be run, the service actor does not start the application as a local process in its own execution context as it would with a desktop application. Instead, the actor modifies the web application’s files by accessing the local file system and inserting code as defined by the actor’s logic. The actor then starts a web server on some physical address that serves the modified JavaScript, HTML, CSS files, images, and other resources to the World Wide Web. Finally, the service actor sends a response to the client with the physical location of the web server so the client can download the modified files. The client is then responsible for starting an execution thread that can access the web server address, such as by running a browser.
Figure 3.4 – Clients send code to be incorporated into the application to actor $a$. The way code is incorporated is dependent on how actor $a$ interprets messages. Actor $a$ translates the closure it receives into code that can be executed by the application and determines which application files should be updated. The actor then starts a web server and sends the IP address and port of the web server for clients to access. Client actors open up a browser in order to access the modified application.

The service actor could start the web server before modifying any files as well. However, if the actor chooses to make copies of the files in order to modify those copies instead of modifying the original files, then the web server won’t know to serve the newly created files. In that case, the web server would either have to be restarted or another web server would have to be created.

We can take the basic web application example and extend it to be applied to a full service consisting of a multitude of applications across a variety of different devices. The principle is the same, allowing clients to modify resources, however this time each message receiver on the service’s island is designed to modify one of its many applications. The service’s island can be set up such that each actor handles a different application’s GUI, or a single actor can interpret messages in a way that lets it handle adaptations for multiple GUIs. For example, a service actor could control two separate applications, or two actors.
can each control one application. Figure 3.5 demonstrates the latter where actor \( a \) modifies the desktop version of the application and actor \( b \) modifies the mobile version.

![Diagram of service with two applications and two clients](image)

Figure 3.5 - Similar to Figure 3.4, but now the service contains two actors that service different web applications.

Where does a client acquire a CURL for a service it would like to adapt? There are several ways to approach this depending on how secure a service needs to be. Our technique uses a global keystore located on a file system that can be accessed by all islands. Actors publish their CURLs to this keystore for other islands to use. Although the way in which CURL distribution is performed falls outside of the scope of our presented technique, it should not impede the practical uses of this method for creating adaptive GUIs. In our demonstration we assume that the clients and service fall within the same organization. As such, global access to the keystore is sufficient. In practice this keystore can be protected such that select, trusted clients can access the published CURLs.
3.3 Justification

It is impossible to predict all the needs of every client for a service for all time. Application designers can only anticipate so much until some new piece of functionality is required or some form factor that was not taken into account surfaces. At the rate at which new devices enter the market it is difficult to make sure that those devices will be appropriately serviced. Our approach reverses the need for an application to adapt to new devices by allowing stakeholders of new devices to adapt an existing service. Application designers must still anticipate needs but only in a way that determines the adaptability of the service itself. As an example, it would have been difficult to anticipate creating a GUI that is dynamic enough to fit a watch face back in the early days of personal computing. It would not have been as difficult, however, to anticipate the fact that new devices may want to modify the graphical presentation to fit various screen sizes or to modify the interactions possible with an interface and thus allow adaptations that can allow for a dynamic GUI that can work with screens the size of a watch or other screen sizes.

Creating a GUI that is infinitely flexible is also nearly impossible. While flexibility is encouraged these days with the help of tools such as layout managers and responsive design principles, these tools begin to break down in extreme cases where the viewport is simply not large enough or the form factor of the devices varies greatly from predicted use cases. For example, responsive CSS [21] works well when dealing with a multitude of screen window sizes but begins to burden the software design once greater changes to the HTML structure are necessary to accommodate a particular device. This burden manifests itself as coupling between the code written for each granular window state. As a side-effect,
changes that need to be made to code affecting a particular discrete screen size could end up impacting other screen sizes as well.

A client interested in increased or reduced functionality of a service requires that the application itself be updated. COAST’s mobile code properties allow functionality to be introduced or removed in a safe way that affects those clients that serve to benefit the most while leaving other clients unaffected. It also allows for a level of customization that’s granular enough to allow two similar clients to adapt the capabilities of the same application for servicing different needs. In effect, the changes made by one client do not need to affect other clients and require no updates on the part of the other clients.

Many existing approaches to building dynamic GUIs for services and applications require foresight on the part of the developer to make sure the needs of devices with varying form factors are satisfied. By anticipating the types of clients that might use a GUI, developers can attempt to design those GUIs with a level of flexibility that they predict will be best handled by the common denominator of all clients that the developers are willing to support. Our adaptation technique breaks from this approach by allowing the clients themselves to make modifications to an existing service or application to suit their needs. With this approach, a GUI developer need not anticipate certain device types and can instead concentrate on broader design concerns such as the ability to allow the GUI to be resized or the ability to allow the GUI to be extended to have additional components without worrying about what those sizes must be or what those components might be.
3.4 Limitations

Our approach strikes a balance between allowing clients to adapt a service’s GUI and a service protecting its internal resources from abuse. COAST provides inherent security properties but cannot guarantee that flaws in the design won’t open up vectors of attack to internal resources should those resources be made to be accessed. For example, one client might remove functionality from a GUI, and through poor design of the application, that modification could affect the GUI for other clients as well by overriding a file on disk. This attack vector cannot be created solely on the client’s end, so it’s important to test the internal functions of any application to make sure they do not create any unintended side-effects of running any computation. It is up to the designer or architect to determine the boundaries of what resources can be accessed and modified as well as how those resources can be accessed and modified.

Applications designed with this technique do not allow clients to directly discover, modify, or access system resources used by those applications. This is enforced by COAST’s security properties that prevent attack vectors associated with mobile code as well as by disallowing direct communication between the application and clients. In order for a client to make a modification to a resource controlled by the application, the service actor has to be designed to allow messages that are interpreted in a way that ultimately leads to the modification of those resources. As such, it is not possible for a client to define a new way for a service to modify its own resources. This makes it so that clients cannot make unintentional modifications to resources and potentially disrupt the service and any applications it may host for other clients.
The COAST architectural implementation requires messages sent via a CURL to be serialized and deserialized by an island’s actor. Certain constructs, however, are not currently serializable at this time, either because MOTILE does not support the serialization or the Racket language does not support it. Such constructs include Racket threads, struct types outside of structs defined by MOTILE, and most non-basic types including recursive types and union types. Any closure defined by a client that contains any of those constructs cannot be sent in a CURL. This brings on additional technical limitations in the kinds of capabilities that can be defined by a client to then be executed on a service. Figure 3.6 defines a lambda that would install a new actor onto an island as a Spawn command. The lambda includes logic that the actor would perform, including the ability to create a web server that runs on a physical address located on the island. MOTILE has no means of serializing the Racket thread or Racket servlet definitions at this time, and so such a closure cannot be sent as part of a MOTILE message. Although this places a limit on the kinds of messages and adaptations that could be introduced to an application using MOTILE, this has the benefit of disallowing code that could be used to obtain access to system resources in some way.
Due to our use of MOTILE, only systems that can run Racket code can currently use our technique for their applications. COAST is an architectural style that is language-agnostic, but no implementations in other languages exist as of this writing. Additionally, any clients wishing to communicate using MOTILE must also be able to run Racket code. This limits the ecosystem in which our technique can exist for now. It’s important to note that this limitation does not affect end-users the same way. End-users do not need to run Racket code and are only limited to whether or not they can run the application’s native code, which can be written in any language.

Clients are further limited by their ability to understand the underlying programming language of a GUI application in order to make meaningful modifications to it via mobile code. Alternatively, they must use a programming language in their mobile code that the receiving actor can then translate into the application’s native code. This limitation should be specified and fully described as part of an API or other discoverable description.
CHAPTER 4

DEMONSTRATION

To demonstrate how to implement our technique for creating adaptive GUIs, we present COASTCalc, a browser-based four function calculator. COASTCalc consists of two parts, a web application that end users can access to use the calculator and a MOTILE actor that can receive messages from clients in order to adapt the application's GUI or to modify its functionality in a way that requires display modifications. In this chapter we demonstrate how both the actor and the web application are built, detailing the design decisions that are made to determine the adaptability of the application and what aspects of COASTCalc can be adapted. We also discuss the implementation of three different clients that define adaptations to be made to the COASTCalc application's GUI. Finally, we tie the components together in one unifying architecture, showing how MOTILE's implementation of COAST is used to enable communication between each client and COASTCalc such that those clients can effectively send adaptations for COASTCalc to incorporate into its application.

4.1 COASTCalc Web Application

The COASTCalc web application is built using JavaScript, HTML and CSS. JavaScript defines the business logic required for the calculator to function, HTML tags are interpreted by a browser to render GUI components and CSS attributes are applied to those tags in order to provide styling and presentation information to the HTML. The calculator as
pictured in Figure 4.1 supports the digits 0-9 and has mathematical functions allowing for addition, subtraction, multiplication, division, and taking the exponential of all real numbers. A text box allows for inputs by typing out numbers or by hitting the appropriate numbered buttons. It also serves to display the latest result of a mathematical calculation.

![JavaScript calculator](image)

**Mathematical calculator**

**Input any number with up to 6 fractional digits:** 

<table>
<thead>
<tr>
<th>7</th>
<th>8</th>
<th>9</th>
<th>C</th>
<th>AC</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
<td>+</td>
<td>/</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
<td>+</td>
<td>-</td>
</tr>
<tr>
<td>+/-</td>
<td>0</td>
<td>.</td>
<td>=</td>
<td>EXP</td>
</tr>
</tbody>
</table>

Figure 4.1 – The COASTCalc web application as displayed in a browser

The calculator business logic utilizes the Dojo Toolkit [30] as its framework for the creation and instantiation of UI components. Dojo provides not just libraries for creating UI components but also other utilities for building web applications in a structured manner. We use Dijit [30], Dojo’s UI library to define and instantiate the various UI components found in the application. UI components are defined and instantiated inside of a Dojo ready function, which is executed after the page is loaded. This function uses imported components of the Dijit widget library, namely the BorderContainer, ContentPane, Button, and NumberTextBox which are required for the creation of the calculator’s layout, its buttons, and its input field. Dojo takes care of binding these components to HTML tags that are defined in the page markup, tying the declared functionality to the rendered GUI component.
In Figure 4.2 above we instantiate a button representing the numeral seven. It takes as parameters a label, an onClick function, and the id “seven”. The label is what gets displayed on the button in the rendered GUI, the onClick function is fired when the button is pressed, and the id is used by Dojo to bind the declaration of the button with a matching HTML tag with the same id. The onClick function in this case executes another function AddDigit, shown in Figure 4.3, which is defined outside of the Dojo ready function.

dojo.ready(function(){
    var button7 = new dijit.form.Button({
        label: " 7  ",
        onClick: function()
        {// Do something:
            AddDigit("7");
        }
    }, "seven");
})
function AddDigit(dig)
{
    if (Current.indexOf('!') == -1) //if not already an error
    {
        if (eval(Current) == 0 && Current.indexOf('.') == -1))
        {
            Current = dig;
        }
    }
    else
    {
        Current = Current + dig;
    }
    Current = Current.toLowerCase(); //FORCE LOWER CASE
}
else
{
    Current = " Hint! Press 'AC'"; //Help out, if error present.
}
if (Current.indexOf('e') != -1)
{
    var epos = Current.indexOf("e");
    Current = Current.substring(0,epos+1) + Current.substring(epos+2);
}
if (Current.length > MAXLENGTH)
{
    Current = "Aargh! Too long!; //don't allow over MAXLENGTH digits before "."
};
document.Calculator.Display.value = Current;

Figure 4.3 – The AddDigit function is responsible for appending a digit to the current value of the calculator.

The AddDigit function demonstrates the logic necessary for processing numerals when pressed. Besides handling error cases such as when the number of digits exceeds MAXLENGTH, the function determines if the number should replace any current value stored in the calculator's memory as defined by the variable “Current” or if the number should be appended to the current value. It is also responsible for updating the display of the calculator's input field by setting the display's value to “Current”. Again, the Dojo binding allows this change to be reflected in the rendered UI component.
The calculator business logic includes both mathematical and non-mathematical functions. Additional non-mathematical functions required for the calculator’s operation include Dot(), DoExponent(), PlusMinus(), Clear(), AllClear(), and Operate(). Dot() allows the calculator to use decimals, DoExponent() allows the user to input an exponent, PlusMinus() allows the negation of the current value, Clear() resets the current value, and AllClear() resets the current value and any stored values. Operate() as shown in Figure 4.4 executes Operation, which is itself a variable that stores a mathematical function to be executed when the user presses the equals button. The mathematical functions as defined in the basic calculator are add(num1, num2), subtract(num1, num2), multiply(num1, num2), divide(num1, num2), and exp(exponent). The variable Operation is set when a button is pressed corresponding to a valid mathematical function. This allows Operate() to be defined in a way that is agnostic to the mathematical operation being performed. This becomes important later when a new mathematical function is introduced that the system is unaware of, and the decision to make Operate function-agnostic enables not-previously defined mathematical functions to be added as working adaptations.
function Operate() {
  Current = Operation(eval(Memory), eval(Current));
  Operation = φ;
  Memory = "0";
  Current = Current + ";
  if (Current.indexOf("Infinity") != -1) // eg "1e320" * 1
  {
    Current = "Aargh! Value too big";
  }
  if (Current.indexOf("NaN") != -1) // eg "1e320" / "1e320"
  {
    Current = "Aargh! I don't understand";
  }
  document.Calculator.Display.value = Current;
}

Figure 4.4 – The Operate function executes the function held within Operation based on what is stored in the calculator’s memory as well as the calculator’s current value. It then sets the value of the calculator to the result.

An additional function addOperation(op, id) shown in Figure 4.5 allows for the introduction of new mathematical operations to the calculator. It works by first creating an HTML button element, setting its id to the passed in id parameter, and attaching that button to the DOM by looking for the placeholder element ‘newrow’ to determine the proper insertion location. Lastly, a Dijit button is instantiated whose onClick function is defined as op, the operation passed in as a parameter to addOperation and whose id matches the HTML element. This function is run by JavaScript code that is introduced to the application from a MOTILE actor outside the scope of the web application itself.
Figure 4.5 - AddOperation adds additional buttons to the calculator that execute new mathematical operations.

```javascript
function AddOperation(op, id) {
    //create the button with the given id
    var row = document.getElementById("newrow");
    var column = document.createElement("td");
    var button = document.createElement("button");
    button.id = id;

    column.appendChild(button);
    row.appendChild(column);
    var widget = new dijit.form.Button({
        label: id,
        onClick: function() {
            // Do something:
            Operation = op;
            Memory = Current;
            Current = "";
        }
    }, id); //options,elementID
}
```

The HTML markup of the application contains tags for all of the basic calculator functionality that comes by default as well as placeholders made up of HTML tags and comments that are effectively ignored by browser rendering engines. In Figure 4.6 we see a snippet of COASTCalc's markup showing the location of the 'newrow' placeholder that provides guidance for the insertion of new buttons to the presentation. These placeholders serve not just for the addition of new mathematical functions but also for additional scripts and styles that allow new functionality and styling attributes to be introduced. The locations of these placeholders are a result of design decisions that allow the calculator to add new buttons in an organized fashion and to allow changes to the presentation of the calculator’s GUI components.
4.2 COASTCalc Actor

COASTCalc consists of a MOTILE actor *Alice* written in Racket that is responsible both for sending and receiving messages and for creating one or more web servers that end-users can access to download and run the COASTCalc web application. *Alice* accepts incoming messages from other MOTILE actors that hold an appropriate CURL with which to communicate with *Alice*. It is this actor that allows clients to modify the web application to meet their individualized needs as clients do not directly modify or have access to the COASTCalc web application itself. *Alice* acts as an intermediary in order to apply adaptations correctly and to enforce the scope of adaptability of the GUI. As described in

![HTML markup snippet](Figure 4.6 - A snippet of the HTML markup that define the mathematical operations that exist as part of the basic functionality of the calculator)
chapter 3, clients send REMOTE messages to COASTCalc and receive a REMOTE message response. Figure 4.7 shows the flow of communication between the client actor and Alice.

Figure 4.7 – An example of a client actor sending a message to Alice, an actor on the COASTCalc island. Each actor contains a sender and a receiver that can be communicated with only if the actor has the appropriate CURL with which to communicate.

Clients communicate with Alice by sending a message using Alice’s CURL as defined in Figure 4.8. Alice publishes this CURL to a key store that is global to islands created within the same address space. As such, client islands created in the same address space will have access to the key store and thus access to the CURL necessary to communicate with COASTCalc. The CURL itself is generated cryptographically and contains an id, an origin, a path, an access id, and a timestamp for when it was generated.

```plaintext
{define}curl/inline ALICE/CURL/ CALC
#<<!!
SIGNATURE = #"r3QxsXIHTWM_0IpWtur2Wv1Wv1EUJFeNP4ktX1e0Yj1FKIGAFc6qURCF8w1wxCMMDE9bmKSoYyo7Fxh1Cg"
CURL
    id = 988791e6-5896-4fee-8576-326682de2ce6
    origin = #"wvdvN1svfEewM76o5VPKj-4k2fbDhaiTFW61VdUc"
    path = (hello)
    access/id = access:send:hello
    created = "2014-04-30T15:25:57Z"
    metadata = #f
}!!
```

Figure 4.8 – Alice’s CURL, containing a cryptographic signature and an access id that is used by Alice’s transport layer, ensuring only holders of the CURL can communicate with Alice.
COASTCalc is responsible for receiving and interpreting the message, making the appropriate modifications to the web application as defined by Alice's internal logic. Adaptations are made by modifying files pertaining to the web application (namely HTML, JavaScript, and CSS files). After the adaptations are made, Alice creates a web server that serves the modified files and sends a response message to the client with the physical address of that web server. The client receives the message and uses a local web browser to navigate to the physical address.

Incoming messages are continuously listened for by a message receiver defined as a blocking Racket loop by Alice. This message receiver is built using a MOTILE transport layer that defines a set of access controls such as an access id and gates that define the number of times the transport layer can be interacted with. Alice's CURL perfectly match those criteria, and so a message is successfully received if that message uses Alice's CURL for communication. Once the message is accepted, its payload is inspected and checked to see if it fits the required schema defined by Alice's business logic, which is itself represented as a Racket switch similar to Figure 3.1. The messages allowed are structured as triples \(<c, m, \lambda>\) where \(c\) is the client actor's CURL allowing COASTCalc to send a response, \(m\) is metadata containing at least a message identifier, an IP address to be used by the server to accept incoming connections, and a port number on which to serve the COASTCalc application, and \(\lambda\) is an optional lambda containing mobile code to be executed or otherwise integrated into the web application. The message identifiers that are defined by Alice are as follows: \('\text{addFunction}, '\text{addCss}, and '\text{basic}\). Each message identifier may have special requirements for the payload of the message in order for the message to be interpreted correctly. Figure 4.9 shows the flow of logic that is executed depending on the
message identifier that is received. Messages containing unrecognized identifiers are still received if sent using the appropriate CURL but are effectively ignored.

When a message containing the ‘basic identifier is received, Alice knows that the message sender is interested in only the basic functionality of COASTCalc. Since no modifications to the calculator are required by the sender, the lambda parameter in the message is ignored and a web server that serves COASTCalc is created using the IP address and port parameters found in the metadata portion of the message payload. Once the web server is created, a message is sent using the sender's provided CURL with a string representation of
the physical address of the server. Figure 4.10 shows the definition of the web server (a Racket servlet) and the reply that is sent using MOTILE that contains the web address of the created web server.

```racket
{define/contract {serve-browser-ui p}
 (exact-nonnegative-integer? , -> , void)
 (set! BROWSER-PORT p)
 (when (eq? (list-ref payload 3) 'basic)
 (set! BROWSER-PATH "/coastcalc.html")
 )
 (thread (λ () (serve/servlet start
 #:port BROWSER-PORT
 #:servlet-regexp #rx"/foo"
 #:extra-files-paths [list files]
 #:servlet-path BROWSER-PATH
 #:launch-browser? #f
 #:listen-ip [list-ref payload 1]
 ))
 )
 (serve-browser-ui BROWSER-PORT)
 (send (list-ref payload 0) (string-append "http://localhost:" (number->string BROWSER-PORT) BROWSER-PATH))
}
```

Figure 4.10 – A web server is created and listening on BROWSER-PATH and BROWSER-PORT. The location of that web server is sent in a response message.

When a message containing the ‘addFunction identifier is received, Alice executes a code path that reads the Racket function found in the message’s lambda, compiles it into JavaScript, and then stores that generated JavaScript code into a new file that can be imported directly by COASTCalc. In Figure 4.11, Alice first retrieves the name of the function from the message. Next, a Racket file is generated that contains the mathematical function defined in the lambda portion of the message. A library called Whalesong [31] compiles that Racket file and pipes the output into a JavaScript file. Whalesong is also invoked to generate a runtime JavaScript file that includes the Whalesong library such that any Whalesong generated files can be interpreted correctly.
Figure 4.11 – Creating Whalesong generated JavaScript files that include the Whalesong library and the compiled Racket function.

Once the JavaScript files have been created, those scripts are added to the COASTCalc application by inserting HTML script tags whose sources correspond to the generated files. Since Whalesong has its own namespace for its variables, and because those variables store information about the function we’re interested in adding to COASTCalc, bridging code is required to in order to use the data in those variables. Figure 4.12 demonstrates the bridging code needed to look up the mathematical function scoped within the Whalesong runtime library and stores it as a JavaScript function locally. This bridging code also creates the code needed to make a call to AddOperation(func), which is already previously defined by COASTCalc as the means to add a new mathematical function to the calculator.

Figure 4.12 – Bridging code to connect Whalesong namespaced functions with the existing JavaScript
All of the script tags and bridging code are inserted based on comments found in the HTML source of COASTCalc. This ensures that the scripts and code are placed in appropriate places in the HTML markup and prevents issues where a prerequisite script is loaded after another script that required the prerequisite script to be loaded. After the inserts are made, a new file is created and saved onto the local filesystem, preserving the original COASTCalc application while creating a new one that can be used by the interested client. After the new COASTCalc files are created, a web server is started as with a ‘basic message, except that it instead serves the newly created COASTCalc application. The physical address of this web server is sent in a response message to the original message sender.

When a message containing the ‘addCSS identifier is received, Alice executes a code path that attempts to insert new CSS definitions into the existing COASTCalc CSS. In Figure 4.13 Alice reads the CSS code found in the message’s lambda and creates a CSS file containing that code. A link tag with the stylesheet is appended to the HTML at the placeholder ‘<!—inject scripts here →’. The new HTML markup is then saved into a new HTML file called coastcalc-mobile.html that is served by the web server that is created. In this case no execution of the CSS is done by Alice as the code is defined as a string and can be written to file as-is.
Figure 4.13 – First we create a CSS file from the payload and then inject a link tag with that CSS into COASTCalc’s HTML, saving that HTML as coastcalc-mobile.html. Then a web server’s path is configured to point to the new COASTCalc application.

4.3 Client Implementations

In our demonstration we create three clients that each represent a different set of user's needs for COASTCalc. Client 1 is a desktop client that is not interested in making any adaptations to COASTCalc and whose end-users want to use COASTCalc as-is. Client 2 is another desktop client that represents users who require the use of the mathematical function modulus, a function that does not exist in the base implementation of COASTCalc. Client 3 is a smartphone client whose users are interested in using COASTCalc from a mobile phone and require that the view of the application be better suited to a touch-based interface.

In chapter 3 we show that we can have multiple client actors all communicate with a single actor or each with a different actor. Because we are adapting a single application for different uses, we have consolidated the message handling logic into one actor that manipulates the web application's files directly by modifying and saving those modified
files onto its local filesystem. Figure 4.14 shows the communication between COASTCalc’s actor *Alice* and three different client actors.

For our demonstration, in order for any of the clients to communicate with COASTCalc, they first need a CURL to communicate with *Alice*. Since *Alice* has its CURL published in a key store in a locally accessible address space, we chose to create each client island within the same local address space. As such, each client’s actor can also publish their CURLs to the same key store. All actors can access any CURLs published to the key store, thus allowing any actor to communicate with COASTCalc. In practice, these clients do not need to exist in the same local address space if the CURL is acquired via other means,
such as a CURL authority (similar to a certificate authority) or via an offline agreement. In order to simplify the demonstration, we also have each client open a browser, representing a sample user that would otherwise consume COASTCalc since the web server exists on localhost. However, once the web server is created any device connected to the local network who has the IP address of the web server can access the COASTCalc application through a browser.

4.3.1 Basic COASTCalc Usage

Client 1 is interested in giving its end-users access to COASTCalc without any modifications. Client 1’s island consists of one actor, Bob, whose purpose is to communicate with COASTCalc. The island holds a CURL pertaining to Alice and allows Bob to communicate with Alice, and thus COASTCalc. Bob initializes a loop with a MOTILE transport layer, allowing Bob to send and receive MOTILE messages. Bob creates a new message, sending a formatted tuple represented by a Racket list shown in Figure 4.15 to COASTCalc. The message contains the following payload: A CURL needed to communicate with Bob, an IP address, a port number, and the symbol 'basic. The lambda parameter in the message is not set because it is effectively ignored by Alice anyway, although it could have been included if the message sender is unsure about the implementation of the message receiver.

```
(send u (list u/reply "127.0.0.1" 1337 'basic))
```

Figure 4.15 – Bob's message sent with COASTCalc's CURL, defined as u. Bob's CURL is defined as u/reply.

After Alice receives and interprets the message as described earlier, Alice uses the CURL found in the message to reply to the sender the physical address of the web server.
When Bob receives the physical address of the web server created by Alice, Bob opens a browser window and navigates to the address. The end-user now has access to the COASTCalc web application.

### 4.3.2 Adapting COASTCalc Functionality

Client 2 wants to adapt COASTCalc to include an additional mathematical function, modulus. Client 2’s island consists of one actor, Carol, whose purpose is to not only communicate with COASTCalc but also to define the modulus function. In Figure 4.16 we show that Carol defines the modulus function as a Racket lambda that can be interpreted by Alice.

```racket
(define func '(((provide mod) (define (mod x y)
 (modulo x y)))))
```

Figure 4.16 – Modulus definition in Racket as defined by Carol.

Actor Carol holds COASTCalc’s CURL in order to communicate with it. Like actor Bob, Carol defines a similar transport layer containing a message receiver and sender. After the message receiver is set up to listen to incoming messages, Carol sends the formatted message in Figure 4.17 to COASTCalc. The message contains Carol’s CURL so that Alice can send a response, an IP address, a port number, the symbol ’addFunction, and a closure func containing the modulus lambda from Figure 4.16.

```racket
(send u (list u/reply "127.0.0.1" 1234 'addFunction func))
```

Figure 4.17 – Carol’s message sent with COASTCalc’s CURL, defined as u. Carol’s CURL is defined as u/reply.

Just as with client 1, client 2 will also receive a reply with a physical address with the location of a newly created web server from COASTCalc. This address serves the newly
adapted COASTCalc application with the ability to calculate modulus as shown in Figure 4.18.

![Figure 4.18 – COASTCalc adapted with client 2’s modification, a usable modulus function.]

4.3.3 Adapting COASTCalc Styling

Client 3 wants to change COASTCalc’s styling because its end-users are on mobile devices and would prefer a better experience for tablets and smartphones. To do this, client 3 defines CSS that is best suited for the mobile experience and sends it to COASTCalc. Client 3’s island consists of a single actor, Dave, which is responsible for defining the CSS needed to make COASTCalc mobile friendly and sending it in a message to COASTCalc. Dave defines the CSS modifications as the Racket string shown in Figure 4.19. The string is a representation of valid CSS that modifies the presentation of various HTML elements to make those elements more mobile-friendly. Specifically, it makes the width of the element containing all of the buttons 100%, allowing the buttons to fill up and use all of the available space rather than staying confined to a small section of the screen. The modifications also remove a lot of the text that would otherwise take up a lot of screen real estate. Lastly, the buttons are made to be larger, allowing them to be more easily pressed by a finger.

```racket
(define mobile-cs "#Display { height: 50px; font-size: 36px; } \ 
label[for=Display] { display: none; } .wrapper { width: 100%; } \ 
span.dijitReset.dijitInline.dijitButtonNode { height: 50px; font-size: 40px; }")
```

Figure 4.19 – Dave's CSS modifications defined
Just as Carol and Bob, Dave holds Alice's CURL in order to communicate with COASTCalc. After Dave defines the CSS, it sends a formatted message as shown in Figure 4.20 to COASTCalc. The message contains Dave's CURL in order to receive replies, an IP address, a port number, the symbol 'addCSS, and a lambda mobile-css containing the CSS definition.

```
(send u (list u/reply "127.0.0.1" 2345 'addCss mobile-css))
```

Figure 4.20 – Dave's message sent with COASTCalc's CURL, defined as u. Dave's CURL is defined as u/reply. The symbol mobile-css is a lambda containing the CSS definition as a string.

The web server procedure mirrors that of the other two clients. Dave is able to access the web server by navigating to the physical address in a web browser, displaying the modified COASTCalc application shown in Figure 4.21.

Figure 4.21 – A side by side comparison of unmodified COASTCalc (left) and Dave's custom-tailored version of COASTCalc (right) after applying Dave's CSS modifications.
CHAPTER 5

ANALYSIS

We have demonstrated how we can use mobile code powered by MOTILE and the COAST architecture style to create adaptive graphical user interfaces that can be modified by clients. In this chapter we’ll discuss how client adaptations can be scoped by software developers to anticipate client needs as well as to prevent possible malicious mobile code attacks. We’ll compare this technique with traditional approaches for adapting an application to suit clients, such as a straightforward API and building out the application. Last, we explain how the COAST architectural style facilitates the implementation of our technique.

5.1 Client Adaptations

It can be difficult to anticipate the needs of every device when developing an application, especially when new devices and software are constantly entering the market. In order to build an application to meet the GUI demands of all of these different devices in a scalable manner, we allow stakeholders interested in supporting certain devices to define adaptations that an application applies. An application developer then no longer needs to anticipate every individual device that may want to interact with the application but rather only needs to create broader rules for adaptations that are allowed to take place. For example, instead of building an application that works on desktop computers and mobile screen sizes, a developer can instead allow an application to be adapted for any screen size, allowing a third-party client to specify what modifications are required to best support its
devices. As another example, a developer might implement the placement of GUI controls such that clients can replace those GUI controls in the application with ones that provide a better user experience for devices that have unanticipated form factors. However, the scope of allowed adaptations still needs to be determined by the application developer in order to prevent a client from potentially abusing the resources used by the application.

Because clients use mobile code to send adaptations to an application, developers using our technique must scope what changes an adaptation can make in order to prevent the accidental access of resources by client code. This is important because we never analyze the mobile code itself to determine if the code is malicious. If we assume that any mobile code received by clients could attempt to exploit an application’s resources, then any scope defined by the developer has to make sure that resources (e.g. a web server, the filesystem, the application’s running memory) are not accessible by the mobile code either directly or indirectly. In order to prevent the access of resources, a developer should make sure that any mobile code received by the application is run in its own execution context such that it has no ability to make function calls that access external resources.

In our demonstration, clients are fully aware of the capabilities of the application and can make adaptations to the GUI based on knowledge about the internal structure of the GUI’s components. Because CSS is downloaded to a user’s device, all the code that defines the presentation of the layout of the view can be inspected, allowing a client to more easily create a message that manipulates that presentation. Another GUI application may not have its code as exposed, so an API or documentation would be required for a client to create a meaningful message to send to the service.
Simply holding a CURL allowing for communication with an application actor does not provide any indication about what kind of message the actor expects to receive or how that actor will interpret messages. COASTCalc has three capabilities, the ability to allow clients to consume the calculator application as is, the ability for clients to modify the CSS of the application, and the ability for clients to add new mathematical functions to the calculator. COASTCalc’s CURL definition does not contain information about any of those capabilities, so clients had to have prior knowledge about how to structure messages and how to create mobile code that COASTCalc can successfully execute and integrate into its application.

It may not be necessary for a client to know the underlying language implementation to effectively adapt a GUI. In our demonstration, we showed a client that wrote a modulus function in Racket and sent a message with a closure containing that function. That function was compiled into JavaScript and then inserted into the JavaScript code of the application. The application was built at an appropriate level of abstraction to allow such an adaptation to occur without exposing the full implementation to the client nor requiring the client to write any JavaScript. This approach still comes with limitations however, since the modulus function had to have been created as a Racket function due to the requirements of COASTCalc’s actor. Otherwise, the message would have been misinterpreted. Support for additional languages lies at the hands of the application developer.

5.2 Application Adaptability

In-house development is by far the most prominent way of maintaining and updating software applications. The problem with this method of software development is
that it requires more and more developers and/or more and more teams to support all of the ecosystems and devices an application should support. It also leaves potential gaps in device support if the application is not tested on every single device the application might run on. One way to ensure that a device optimally supports an application is to allow some stakeholder of that device (a client) to define the needs of that device. A client is potentially able to make the best decisions in terms of providing new functionality and a suitable plan for changes to the application's GUI that is consistent with the functionality being provided.

By allowing the implementation of new functionality by clients rather than by the application developers, a greater range of potential features and GUI adaptations for clients is possible. Application developers can focus on scoping abstractions and focusing on non-functional requirements that allow their applications to be more or less adaptive, affecting a client's ability to make modifications without compromising those applications. With our technique, developers and software architects can scope adaptations for an application by choosing the appropriate level of abstraction for which messages are interpreted by one or more application actors. These decisions ultimately affect the types of messages that clients can send and with that the scope of the adaptations that are allowed to occur. This allows developers a means to ensure that a system that allows mobile code to be executed and incorporated into its existing environment does not cause system instability or cause the system to start incorporating functionality that goes out of scope of the application's intended offering. COASTCalc should probably not, for example, suddenly begin to execute services which control human vitals nor should the basic calculator functionality be completely disabled. Our technique does not explicitly prohibit either of those scenarios.
however should an application or service actively wish to expand or limit their offerings in such a manner.

A benefit of allowing clients to introduce GUI modifications and new functionality is that an application can support future devices even while the application is still being developed. To handle new devices, an application developer anticipates the sorts of adaptations that clients might need to make, and then scopes the actor’s message handler to appropriately handle those kinds of adaptations. The developer is also responsible for making the application robust enough to accept the introduction of changes from client-defined adaptations. For example, COASTCalc was scoped in a manner to allow messages that defined new CSS without having to know anything about what a client’s specific changes are. In this specific case, a client took advantage of the CSS adaptation and used it in order to manipulate the GUI styling to best fit the screen of the client’s device. The application was built to be dynamic enough to accommodate new CSS styling in order to complete the modifications.

We can directly compare this method of introducing presentation changes to what is more prevalent today, responsive CSS. Responsive CSS uses CSS3 media queries in order to allow a single style sheet to specify the presentation of the layout for discrete screen widths [30]. The developer must explicitly define each screen size for this approach to be effective. If the layout is ineffective at a screen size that was not anticipated, the presentation of the GUI becomes suboptimal at that screen size. Our approach avoids this issue entirely by allowing any client of that screen size to send a message containing an adaptation for that screen size. The two approaches are not mutually exclusive however.
With COASTCalc, clients can send messages containing media queries that are appended to the existing style sheet that define new rules for specific screen sizes.

In order to make more drastic changes to the GUI of COASTCalc we would have to alter COASTCalc's actor to grant greater degrees of freedom for the scope of adaptation allowed. For example, if a client wished to change the existing addition function on COASTCalc, we would need to allow for a closure to be interpreted in such a way that it affects the existing math functions of the application. Done correctly, such an adaptation would blow open all sorts of broader, impactful changes that other clients may also want to perform, such as removing the basic math functions entirely and replacing them with different math functions. However, choosing the degree of freedom for adaptations and dynamicity is up to the application developers and has implications for how far a system can be extended and for the architectural quality of the system [31].

5.3 Client Ecosystems

It is possible to allow different clients to make cumulative changes to a GUI without ever needing to know about each other's requirements or the modifications each has independently introduced to the application. How the responsibility of each client is determined in the adaptation of GUI components is entirely up to the ecosystem itself. It's possible for business arrangements between clients and application developers or even among application developers to dictate how different actors interact to provide a GUI service that clients can use effectively.

Not all clients need necessarily have access to the same kinds of adaptations allowed by an application. There may be certain demographics of clients that are interested in
modifications to an existing application that remain private and inaccessible to other clients or even modifications that require a specific kind of adaptation that is not generally allowed by the application. Third parties can make special arrangements to allow custom and possibly private modifications. An application can contain several actors each with their own CURLs that provide different kinds of adaptations. Each actor would contain different capabilities that provide greater or lesser degrees of freedom for the adaptation of the system, allow access to components otherwise unavailable, or allow specialized access to data or system resources. The CURLs for access to actors that provide broader mechanisms for application adaptation can be sold as a premium feature for paying clients or as part of an exclusive business deal.

5.4 Using COAST for Adaptive GUIs

Our demo uses MOTILE, an implementation of the COAST architectural style as our base framework; however, we also use a hybrid of component/object architecture as well as a scripting language approach [31] which allows components defined in mobile code by service consumers to be installed on the application. How higher level components are designed is left entirely up to the application developers, allowing those developers the freedom to use COAST as part of their overall architectural solution in a flexible way. For COASTCalc, our implementations could have used an API approach or a plug-in approach at a higher level to meet our communication goals between client actors and COASTCalc’s actor, since both approaches are popular with many of today’s existing services. The ability to call an API or to install a plugin and have a service or application give back some GUI representation of the plugin or result of the API call is not new. Browsers today can
communicate with services via URLs and receive mobile code (HTML and JavaScript) in the response which is then rendered appropriately by those browsers.

In our approach we demonstrate the ability to go in the opposite direction, sending code from a consumer to a service provider in order for the service to provide a custom experience for that consumer. Although this is also not a new idea, as consumers and providers will often communicate back and forth during a session in order to provide the consumer with a meaningful experience, we differentiate ourselves by having the whole messaging and distributed infrastructure as the underlying architecture rather than components built on top of some other architecture.

So then why use COAST at all if all of these ideas can and are already implemented in the real world? The answer lies within the properties of the COAST architecture itself and its guarantees. There are too many ways to compromise the security, the integrity, or even the trustworthiness of a system by having to manually tackle those nonfunctional requirements explicitly. These concerns are simplified from a high-level perspective when the underlying architecture, implemented correctly, provides guarantees about the properties of the system. Thus, a lot of the pain points in building distributed services and systems serving GUI-based applications are removed by taking advantage of COAST's axioms at the architecture-level.

- “All services are computations whose sole means of interaction is the asynchronous messaging of closures (functions plus their lexical-scope bindings), continuations (snapshots of execution state) and binding environments (maps of name/value pairs)” [32]
In COASTCalc, a function and accompanying data is wrapped up as a closure and sent as a message asynchronously over a messaging layer defined by MOTILE. So long as an actor possesses a valid CURL denoting the location of an actor within a service, that actor can scope the extent of what a service provider can ultimately provide to that actor. COASTCalc is not just the JavaScript calculator application but also consists of an actor with the ability to augment the application with newly defined math functions and to allow clients the ability to alter the styling of the GUI layout.

- “All computations execute within the confines of some execution site \(<E, B>\) where \(E\) is an execution engine and \(B\) a binding environment ” [32]

Any system using a correct implementation of the COAST architecture will enjoy the inherent security provided by an execution site when executing mobile code. All computations executed are confined to the execution engine in which they are run, whether it be a JavaScript compiler or a Scheme interpreter (both of which are used by our demo), as well as whatever resources the binding environment allows the computations to access. COASTCalc has one actor living on the application island, and that actor has access to particular HTML, JavaScript, and CSS resources living on the file system and nothing else. The client actors have access to a browser context as well as the global JavaScript environment running on the local machine, including any libraries (like Dijit) or other code bound to the global namespace. This prevents running computations from hijacking other resources that can cause system instability and ultimately a security failure on a client machine.

- “All computations are named by Capability URLs (CURLs), an unforgeable, tamper-proof cryptographic structure that conveys the authority to communicate” [32]
This axiom refers to the infrastructure used to send messages between actors running computations. We will not delve into the topic of security because that is beyond the scope of this paper, but suffice it to say that one cannot trust incorporating mobile code sent from an actor outside of one’s system if the trustworthiness of that code cannot be ensured. As such, the fact that communication of mobile code can be claimed as tamper-proof as well as un-forgable allows for a certain degree of trust and authority that is difficult to achieve easily in systems today. For our demo, client actors were given explicit permission to communicate with COASTCalc because they held a private CURL that no other actor could forge.

“This computation \(x\) may deliver a \textit{message} (closure, continuation, or binding environment) to computation \(y\) if and only if \(x\) holds a CURL \(u_y\) of \(y\)” [32]

This concept is not unlike knowing about a particular service on the Internet (denoted by a URL), and then not being able to consume the service due to not knowing the API key necessary for the server to recognize your authorization. However, COAST goes a step further by not even allowing you to communicate with the server at all without having a specific CURL that evaluates to the location of the actor you wish to communicate with. This is most akin to knowing about a service but not knowing the URL to access that service and without being able to accidentally discover that URL unless you’re proven trustworthy. In our COASTCalc demonstration, we published COASTCalc’s CURL in a global keystore that is accessible to all clients running locally. Each client actor (or computation) \(X\) sends data in the form of variables and functions within a closure that is received by actor \(Y\) running on COASTCalc. Had our clients not explicitly been given COASTCalc’s CURL, there would have
been no way for them to communicate because CURLs are unforgeable and thus the authority of an actor to communicate with another actor is also unforgeable.

- “The interpretation of a message delivered to computation \( y \) via CURL \( u_y \) is \( u_y \) – dependent” [32]

For our demo, each message is interpreted based on the contents of the closure contained within the message as well as how the receiving actor has been programmed to handle received messages in its running environment. For each client, the message created contains a closure encapsulating data, and it is up to COASTCalc’s actor to determine how to best handle the contained data. That means that a communicating actor \( A \) cannot be guaranteed that a receiving actor \( B \) will interpret any received data the way that \( A \) would like it to be interpreted. COASTCalc’s actor is the one that has implemented the capability to incorporate new math functions into the calculator and it is the only one capable of interpreting data received to do just that. This ensures that any new code introduced to a computation continues to be interpreted by the original code defined by the computation and is not unintentionally overwritten by any arbitrary code received. Nothing prevents a system, however, from allowing mobile code to overwrite the original programming or augment it in such a way such that the original interpreter running on the actor is lost. It is up to the application developer to define limits as to how mobile code can affect its internal resources, if at all. Nevertheless, this paradigm eases the developer from having to think about how to deal with infinite specific and arbitrary messages that need to be handled by the service with the tradeoff that the developer must now consider how an actor will interpret messages at a higher level of abstraction for running computations.
There are inherent risks when using mobile code and COAST mitigates and even negates many of those risks. In reality any software system can be built out of practically any software or system architecture, but only those systems built from a suitable architecture are likely to withstand requirement changes and maintenance changes that occur over time. Our demo is not meant to prove whether or not the capability to create adaptive GUIs from mobile code exists; rather, they are meant to show the relative ease in creating such systems when using COAST as the underlying architecture for such systems. We demonstrate services that provide distributed services that can be built in a reasonable way and uphold certain principles that would otherwise make building this system difficult, such as security and trust guarantees that prevent the service from failing while still allowing consumers to adapt the application to their specific requirements.
CHAPTER 6

FUTURE WORK

Our demonstration shows basic use cases of the kinds of adaptations clients might make. Although the basic concept of adapting GUIs with mobile code and computations was demonstrated and explored in this paper, a lot more work needs to be done in order to show that this type of architecture and approach works well in other contexts and application domains. In order to test the robustness of our approach, we need to demonstrate GUI adaptations for a greater number of clients with a greater number of needs, we need to create different applications with varying levels of complexity, and we need to demonstrate varying levels of abstraction on application actors. There are many angles that can be taken in order to further study the implications of mobile code in the generation of adaptive GUIs. In this section we present next steps that we believe would be the most worthwhile avenues for advancing the technique.

Currently, the demos run on a Linux desktop platform. To truly test the adaptability and modularity of the GUIs, applications should be built and adapted for a variety of clients running on various hardware and software configurations. One can imagine an experiment consisting of an Android-powered mobile phone, an iPad, a Windows laptop, and the Linux desktop. Each of those clients can run actors that send messages containing information about the capabilities of the devices they are running on as well as functions that a service could use to serve functionality and associated GUI components that would best fit those clients’ needs. Currently we’re limited to using devices that can run the COAST architecture to execute MOTILE code. Future MOTILE compilers may be able to allow MOTILE code to
be executed in a greater variety of contexts, including on mobile operating systems, allowing for a more accurate representation of clients adapting a services’ needs. A future version of the COASTCalc demo could consist of clients running on all sorts of different devices, each communicating with the COASTCalc service running on a Linux machine and making tailored adaptations of the service to work with each of the devices’ form factors and screen sizes.

Another avenue to pursue is to explore varying the degrees of freedom of a GUI by altering the capabilities of the application actor or actors. In COASTCalc, adaptability refers to the robustness of the interpretations of messages received, and this level of adaptability can be altered either by adding new interpretations or expanding/reducing the capabilities of existing execution paths. While we can speak to the advantages and challenges of COASTCalc with its current design, we can merely speculate about the design implications when the application’s adaptability is altered. We could more easily analyze the impact of differing levels of adaptability of the application by comparing varying degrees of freedom with respect to message interpretation and ultimately GUI modifications.

Our current demo only ever serves HTML files that are viewed on a web browser. A fuller example could include various different applications, each with a GUI built using a different technology. A more fleshed-out version of COASTCalc could include both a web application and a desktop application, each with their own GUI implementations that can be adapted by different clients. A desktop client may only ever be interested in the desktop application and makes adaptations to that GUI only.

To explore more practical uses for the creation of adaptive GUls within an ecosystem such as one that may exist between two organizations, we suggest building a
distributed ecosystem for GUI generation. This means that clients could tap into different graphical libraries and other resources that would help to unveil the sorts of challenges developers would face in building such an ecosystem. There are many concerns relating to design, scalability, and security that require further study if we are to recommend COAST as a suitable architectural style for creating those ecosystems. It will also help us to further analyze the challenges in choosing different degrees of adaptation and so we can make recommendations about how system architects should go about determining appropriate abstractions for their software components based on the goals they've chosen for their system. Building out one distributed ecosystem using COAST can also serve as a blueprint for the creation of other distributed COAST ecosystems.
CHAPTER 7

CONCLUSION

This paper demonstrates the creation of adaptive GUIs in an application leveraging the COAST architectural style. Our main motivation for such work is due to the evolution of ubiquitous computing devices which all want to be able to consume the same services and applications, and the fact that developers will have an ever increasing challenge of building GUIs to conform to all of the different hardware and software specifications of these devices. As such, this paper presents a new means of creating adaptive GUIs in applications through the properties of mobile code and the axioms set forth by COAST.

Adaptive user interfaces have been worked on for decades and many different solutions have come about. Each solution brought along its own set of assumptions about the world and its future. We have followed suit by further expanding what the future of computing looks like and by creating our own architectural approach towards adaptive GUI generation that goes beyond an application’s ability to anticipate its different clients and instead allows the clients themselves to define and consume their own tailor-made versions of applications.

We built one application, COASTCalc, to demonstrate a practical office tool application domain that is robust enough to allow adaptations of its application GUI by different clients. The changes clients ask COASTCalc to make show how with the help of an underlying COAST architecture consisting of a higher level component/service architecture, application developers can choose different levels of abstraction which can serve to inhibit or expand the potential growth of their applications. Different concerns
about abstraction and security were discussed as well as guidelines for developers that might adopt such architecture for their systems.

More work with implementing COAST and adaptive GUIs is necessary in order to assess all of the strengths and weaknesses of such an approach against current technologies. A few research avenues are identified, including building out a suite of full-fledged applications that run on multiple platforms as part of an overarching service, building applications in varying programming language that use various GUI frameworks, and taking an application and modifying its degrees of freedom by manipulating the adaptations it allows.
BIBLIOGRAPHY


