REAL TIME ANALYTICS ON DIGITAL DISTRACTION

In Partial Fulfillment

of the Requirements for the Degree

Bachelors of Science in Computer Science

At California Polytechnic Institute of San Luis Obispo

by

Kristian Welch

December 2015

Abstract

Real Time Analytics on Digital Distraction

Kristian Welch

Hypothesis: empirically measuring the users productivity has the potential to significantly increase their productivity. This paper is the documentation of the process of building software capable of verifying this hypothesis. Starting with the research which gives enough evidence to warrant the hypothesis, also included is commentary on the prototyping, design decisions, and iteration based on user feedback of the software.

Acknowledgments

Senior project advisor: Professor Franz. J. Kurfess

Contents

Chapter 1

Introduction

Productivity software is quite popular, bringing in over 62 billion dollars of revenue a year [3]. Automation through software provides a significant return on time invested when judiciously applied. Current solutions offer the user tools such as blocking websites, tracking time, and setting goals. These provide value, but rely on feedback loops that range from minutes to days.

The attempt at a new type of productivity software detailed in this report is called My Digital Referee at the moment. MDR is a shorthand way of referring to the software itself.

1.1 Problem Statement

Productivity stands in contrast to the distractions that most people suffer from. The most cogent example is the most of an hour a day Americans spend on average on Facebook [1]. This is only one of many websites people use, using time that could be spent productively. Even more concerning is that the time

spent on Facebook actually has a negative effect on the user's mind. Studies ranging from today all the way back "to the first HomeNet study in 1998 (link is external), there was a statistically significant relationship between Internet use and depression though the actual cause of that link remains open to debate" [6]. For something over a billion people spend time on regularly, we know surprisingly little about the effects time spent on the internet has. The link between Facebook and depression is correlative in current studies, and by building software that reduces time spent on distractions such as Facebook it may be possible to shed more light on the issue.

1.2 Summary of Current Solutions

"RescueTime helps you understand how you spend your time on the computer by automatically keeping a log of the time you spend on different applications and websites"[7]. My initial approach is modeled off of the same concept of tracking time spent on specific websites and apps in focus on the user's machine. Their feedback is deeply categorized and highly analytical, with the ability to set precise goals. For example the user can set a goal to do 2 hours of writing a day and keep track of that automatically and get notifications. Their feedback and analytics are done server side, meaning they process user data every 30 minutes for free users and 3 minutes for paid users. My personal experience using their software showed value to their approach, but the lack of realtime feedback ultimately made it worth exploring the creation of my own solution.

"Slife makes it impossibly easy to keep track of time and stay productive" [8] Slife also automatically tracks time, and pushes the user to augment this by taking notes about how they spend their time. Daily note taking and interac-

tion is helpful for the most fastidious of users, but there is significant value in simplification of user interaction through automation. The main value of data collection from potential user feedback is that automated collection is where most of the value comes from. User reviews include complaints about the manual part of time tracking. One reviewer stated: "you'll waste more time than you'll save manually finessing the data it collects" [2]

Another popular solution is to block distracting websites and applications. Cold Turkey works on Windows by taking a list of websites and applications the user considers distracting and blocking them. Freedom has similar functionality for OSX, with the ability to shut off the user's internet entirely to help them be productive for set periods of time. This can certainly help, which I can attest to through months of personal use. Unfortunately it is trivial to disable these applications or simply switch to a phone to bypass their blocking efforts.

This represents a brief overview of the functionality of several classes of personal productivity software. Other classes such as to do list software are important but outside the scope of this project for now. This represents a starting point for some of the functionality of the software under construction. They represent hypotheses as well, the first is that recording and displaying the user's time usage to them will result in a moderate improvement in time usage in the range of 10-20% on average. This is an estimate provided simply as a guess to be confirmed or refuted with the empirical data provided by future tests with MDR. The second hypothesis is that blocking will create a minor decrease in time spent on distracting activities at best. It is for this reason there will be an "active control" group that only blocks distractions to serve as a comparison against more novel methods of enhancing productivity.

Chapter 2

Proposed Solution

Like any good development process, developing MDR has been iterative. The following is the proposal from the beginning of the Senior Project, along with a section of changes and improvements made along the way.

2.1 Original Proposal

The goal is to build a service which can help users focus by recording their device usage.The data will be used to give feedback based on features designed with psychological research. My goal was to create an efficient architecture that can be expanded to other platforms with minimal code duplication (OSX, Android, Linux, etc.) The users machine has two main components which run. The local interface is written in the best way to interface with the local operating system (the version being built now is in c# on Windows) and relays the data to the local analytics engine. The analytics engine is a REST api built using golang since it is quite efficient, easy to run servers on, and can be compiled to run on windows, OSX, Linux, android, and more. This adds some complexity,

but is still fairly efficient with its current usage of a fraction of a percent of the users CPU. Golang also made it easy to add a html, css, and javascript based user interface which can be reused between any modern operating systems and web browsers. By using golang much of the code including data models, analytics methods, state management, etc can be reused from operating system to operating system. Golang also is efficient, with its core libraries implementing a web server framework which among the fastest for serving dynamic content based on many publically available benchmarks [9]. It also makes working with JSON fairly simple, which is an incredibly useful technology for storing data and settings due support being native in javascript and based in solid libraries in both C# and golang. Unfortunately traditional SQL databases cannot persist JSON data particularly efficiently which lead me to find a better database for persistence. I put a significant amount of time into finding a better database as I have seen the pitfalls of doing this incorrectly during past experiences at DocuSign and in my own projects. While still somewhat new, aerospike can handle unstructured as well as structured data while having performance that surpasses traditional SQL databases. My research showed that other NOSQL databases such as mongodb, couchbase, and Cassandra were either vastly less performant or more complicated to setup and maintain. There are numerous benchmarks across various workloads showing the advantages of aerospike. In addition, aerospike has excellent documentation to make the setup process simpler. Past experiences at DocuSign using Windows Presentation Foundation (WPF) to build user interfaces showed it to be a fast and effective tool for developing user interfaces. I wanted the UI to work for operating systems beyond my initial validation on Windows. Since html5 based user interfaces work on all modern devices I settled on this despite the increased development time. The UI that displays this data uses Facebooks

reactjs framework to dynamically render html components in a modular fashion. While learning this technology and the other JavaScript frameworks being used took time (JSX, bootstrap, browserify, and more later), the end result involves both gaining valuable new skills and the code reuse between operating systems I sought. The last major piece of this architecture was remote servers to run this on. I put a significant amount of time into researching and testing different methods of hosting. Maintaining my own servers is far too costly in time and money for a project this early in validation ruling that out. Amazon web services is quite simple and popular, but a deeper look revealed slow performance for the price as a cost. Googles cloud based hosting is better, but still costly. My past experiences running servers and analytics in Microsofts Azure made it appealing, but not as attractive when I am paying the still expensive hosting bills myself. Linode on the other hand requires more setup since they simply host linux virtual machines to use however you choose. It took a couple hours to get this online, secured, and verify their performance. This will probably end up being worth the cost as my calculations estimate that $50 a month of linode servers could serve at least 10,000 concurrent users. I now have this set up so that when I have the code tested and ready to run I can simply run scripts to deploy it for a fraction of the cost in the future.

2.2 Modifications

While not rigorously studied, there is evidence showing that half if not more features are rarely used if at all in software [**?**]. Considering the development resources of a student busy with classes and internships in addition to development, developing unwanted features can be a huge setback to the development

process. This realization re-balanced priorities, to the point where Aerospike is no longer a database under consideration. In fact, because the analytics of user data can be done so effectively on the client's machine there is no reason to set up a database to store usage data for the first beta test.

Windows presentation foundation was reconsidered for the UI along the way due to the significant time and effort spent setting up a user interface capable of being operated by any user. Ultimately the setup process appears to have been a front loaded process due to setting up reactjs with a web server that is not nodejs requiring more setup and debugging than expected. According to time logs, just setting up and debugging the libraries and code for the infrastructure to get a usable settings page took over 15 hours. Much of this will be recouped with a more consistent UI style with less future code due to CSS along with a library of reusable reactjs components that will make future pages faster to develop.

Chapter 3

Technical Implementation

The technical implementation of MDR has proven to be a front loaded process so far due to the architectural and design decisions made. This has already paid dividends in making future development simpler and faster.

3.1 Design Decisions

3.1.1 Target Operating System

The most used operating systems on user devices include Windows, Linux, Android, OSX, and IOS. IOS applications do not appear to have an API to do things such as block websites or precisely track the webpage the user is on. This makes implementation of MDR not feasible as of IOS 8. Windows, Linux, Android, and OSX all appear to have the functionality to implement MDR, so it comes down to the utility of implmenting on each of these operating systems. Windows and Android have the largest user bases making them priority candidates. Android may have the advantage of going everywhere in the pockets of

users, but the fact that users spend more time on Windows gives a larger window for improving user productivity. This is contentious and Android is valuable itself, making the recruitment of someone to build and maintain the Android app a future priority.

3.1.2 Analytics engine language

The design constraints here require the analytics engine to be portable accross android, OSX, Windows, and more operating systems if possible. Additionally, running realtime analytics should use a minimal amount of the user's CPU to not impact their machine's performance and especially not battery life. This isn't as much of a problem on Windows, but performance could prove a critical issue when porting to Android. Assessment of build speed and complexity during the development process was another factor for consideration. Java and Scala both had issues with common build systems being more complex and taking up to 10 seconds to build on a Lenovo x240 with an i5 4200U processor and Samsung 840 EVO SSD. Java and Scala also require Java to be installed on the user's machine. On the other hand Golang compiles in less than .1 seconds even was the project has grown. It also outputs a standlone executable which requires no dependencies to run on the user's machine. Besides javascript and JVM based languages, Golang is one of the few which can be used to build Android apps. These factors were among the many reasons for choosing Golang, with the acknowledgement that Java and Scala are viable options that likely would have lowered development productivity in addition to forcing the user to have Java installed.

3.1.3 Windows Controller Language

This was a clear cut choice, as not many languages have bindings to the win32 API calls required to interface with Windows. C++ was less desirable due to lower programmer productivity from dealing with manual memory management. Significantly more experience with C# made it an even easier choice since this made getting productive that much faster on top of having garbage collection.

UI language and framework: Windows Presentation Foundation was considered due to past experience being able to develop user interfaces quicker than using html and javascript. Concerns with this approach include the UI being not portable. It also does not have the styling flexibility that CSS provides to html UIs. Another consideration is that numerous charting libraries exist for web based applications while the options in WPF are far more limited and require more development work to integrate for the current use cases such as displaying a pie chart of how the user spent their time. While WPF would allow for the UI and controller to both be developed in C#, these components do not need to share much code. Developing these components in separate languages enforces their decoupling, simplifying later ports of MDR.

3.2 Architecture

The architecture of MDR on the user's machine has grown to accurately reflect the model view controller design pattern. The model is represented by the analytics engine in golang which holds all user data and settings in addition to handling all state management. The controller is the local component implemented in C# on Windows. It has a predefined set of commands it receives from

the user and the analytics engine to act on. This is not the direct analogue to a traditional controller, which usually implements more logic around issuing commands which exists in the analytics engine to allow greater code reuse in ports of MDR to other operating systems. In MDR the issuing of commands is handled by the golang analytics engine, while the controller also handles collection of user data from the operating system. The view then is fairly clear cut, as it is implemented entirely in javascript, html, and CSS. The analytics engine hosts a local webpage that uses javascript to pull the user's activity data and display it to the user in reactjs components.

While there are discrepancies with the traditional MVC model, such as it usually being implemented in one language and not three, this abstraction has served well. The separation of concerns makes testing significantly easier because each component already has a well defined http based interface to test against. New UI components can be prototyped independently by mocking the json data normally pulled from the analytics engine. The engine communicates entirely with http requests, so unit tests can easily call the exact same interfaces that a user would.

The three part separation of concerns has huge implications for portability as well. OSX will be a comparatively simple operating system to port this application to because the analytics engine and html UI can be completely reused from the Windows version. The only part needed will be a local controller for OSX that complies to the same http interface with the analytics engine that has already been built for Windows. This reusability will likely translate to android as well, since a controller can build built for android in the future. If for some reason the UI proves unusable on android, most of it can be reused by compiling it using react native to substitute native UI components for html ones where they

prove ineffective for the user's needs.

JVM based languages proved appealing due to their ease of portability on all required operating systems.

3.3 Development Process

The agile process makes sense for a consumer based product, but the tighter the cycle your development process the more benefit you get. This is due to the faster customer feedback leading to less unnecessary features being developed.

Planning started with longer cycles of over a month, which worked better earlier on as low level infrastructure was more predictable in scope and requirements. For instance MDR needed a component to record the user's current activity on their computer, so this could be planned in advance. As more of the development process focuses on using built infrastructure to provide specific features to the user, the development cycle has shortened to two weeks to get faster feedback from users.

Backlog planning with Trello has proven better than the initial use of OneNote for tracking current sprint items versus a backlog. Instead of keeping all of these together, separate Trello lists are used to keep track of completed items, current items, and development items for future consideration. Adhering to the backlog allows for more precise criterion for the release criteria of the current sprint.

Sprint execution has varied significantly in cadence due to time allocations fluctuating. From 10 hours a day during thanksgiving break to an hour or two during summer internships and finals week, there were good reasons for this to happen. Implementation follows different patterns in different components too.

Test driven development has proven efficient for development of analytics engine functionality. Where it might take significant amounts of time to test functionality by running the application itself, verifying code accuracy using unit tests takes roughly 120 milliseconds with built in unit testing in golang. Writing the tests first as a definition of expected behavior has already caught and eliminated dozens of potential bugs. Testing for the controller and UI has proven harder, as their functionality would require sizeable testing harnesses and integration tests to fully verify. Instead the tradeoff has been made to keep using all features of the suftware personally and fix bugs in these components using the error logs kept on disk.

Demonstrating sprint functionality has followed the speeding up cadence, with feedback from users at least every two weeks. Feedback is taken into account, creating a list of future improvements to evaluate with more users. As the list of potential improvements grows, the bar gets higher due to constrained development resources. Potential ideas stay listed in OneNote, while improvements with significant validation as well as bugs to into Trello lists for higher priority in the development process. Retrospectives have value as a single developer, but even more so as more team members are added necessitating more communication. This step in the agile process has not been formal, but happens any time enough customer feedback necessitates aggregation into future action.

This agile based cycle has already lead to personally usable software, and will continue to provide an effective method as the cycle shortens. Lengths of several weeks as recommended for using the process will hopefully decrease the risk of wasting large amounts of time developing ineffective or unwanted user features.

Chapter 4

Interface Documentation

The application components communicate via HTTP, using RESTful API endpoints on the analytics engine to interact. The analytics engine is a HTTP server running on the user's machine on port 8191. The following are the endpoints and their functionality.[1]

4.1 /

This endpoint is overridden so that when the webpage localhost:8191 is requested in the browser, the html page with the UI is loaded instead of a JSON response.

[1] The object descriptions are golang code with json mappings.

4.2 /activities

4.2.1 Data Schema

type Activity struct Name string ‘json:”n”‘ DisplayName string ‘json:”d”‘ Description string ‘json:”de”‘

4.2.2 GET

Returns an array of all known Activity objects.

4.3 /classifications

4.3.1 Data Schema

type Classification int Unclassified = 0 Productive = 1 Distracting = 2 Neutral = 3

4.3.2 GET

Returns a map of activity names to their classification value.

POST

Takes an activity and records the mapping from the activity name to it’s classification.

4.4 /records

4.4.1 Data Schema

type ActivityRecord struct Name string ‘json:”n”‘ Duration int ‘json:”t”‘ Classification Classification ‘json:”c”‘

4.4.2 POST

Takes an ActivityRecord from the local controller representing the website or application the user had in focus at the time of recording. Records are posted once a second by the controller.

4.4.3 GET /aggregate

Returns a map of string based names of each ActivityRecord mapped to the sum of the number of seconds this record was posted during the last day.

4.5 /commands

4.5.1 Data Schema

type Command struct Name string ‘json:”n”‘ Args []string ‘json:”a”‘

4.5.2 GET

Returns an of commands generated as a function of the user's data and settings. The controller GETs this endpoint once a second and then runs the commands.

4.6 /setting

4.6.1 Data Schema

type AllSettings struct Activities map[string]Activity ‘json:”a”‘ Classifications map[string]Classification ‘json:”c”‘ SmartOpen []*SmartOpen ‘json:”0”‘ type SmartOpen struct Type ActivityDesignation ‘json:”a”‘ //Application vs website to open Path string ‘json:”u”‘ //uri to open On Days ‘json:”d”‘ //what days to open on Start int ‘json:”s”‘ //seconds from the beginning of the day Window int ‘json:”w”‘ //length of the window of time in seconds DoneLast time.Time ‘json:”l”‘ //tracks when it last ran

4.6.2 GET

Returns the current user settings object stored within MDR.

4.6.3 POST

Overwrites the settings with the new settings object included with the POST request.

Chapter 5

Usability Assessment

Developing MDR is technical in nature, but the most important assessments come in how it translates into software that benefits the user.

5.1 Metrics

What ways can be used to measure user benefit? The initial goal of MDR is to give users productivity gains. The most straightforward metric is how much time the users spend productively. This is built into the core of the product, making it simple to track. So as long as users classify the majority of the websites and applications they use this is as trivial as summing up every user's uploaded activity data. Concurrently, the amount of time the user spends on distracting activities is a metric to collect for analysis.

Discussions with potential users have confirmed the insight that time alone is an insufficient metric. How that time is used matters just as much. For example, 8 hours of productive time can be inefficient if the user is constantly switching tasks

and getting email notifications. The evidence of the costs of multitasking is quite clear, with studies showing that multitasking can cause an effective drop of 10 IQ points on average [**?**]. Another study shows that it can take up to 20 minutes to refocus on a cognitively demanding task if the person performing the task is distracted [**?**]. This is likely due in part to multiple tasks filling working memory, which is a key resource in intelligence as confirmed by IQ research [5]. There are numerous other studies on the subject providing ample evidence that measuring how often the user switches tasks is a metric worth measuring. A more precise version of this can take into account when the user quickly switches between the same two tasks over and over. Similar activites likely have less cognitive load as more of the material in the user's working memory does not have to be switched out each time similar tasks are switched. Therefore weighting the cost of each switch in favor of tasks more recently engaged in should be a more precise metric. This can be easily calculated across a user's data by reaggregating the raw activity data stored server side.

From a more technical standpoint, another metric of importance is the exceptions experienced in the application. The controller and model segments both log their errors to a file on the user's machine for later analysis. These error logs can be aggregated to measure both the raw number and the type of errors encountered. The distinction between number and type is critical, because the application runs large segments of its code once a second. This means that errors involved in the realtime analytics can and already have been logged every second for days on end creating a great deal of noise when analyzing error logs. For this reason the raw number of errors, types of errors, and number of types of errors are distinctly valuable metrics for assessing the software's quality.

User feedback is also important, which is why the taskbar icon has a feed-

back button avaiable up right click. It is also available within the web UI. These both direct to a form where the user can submit text comments on their experiences using MDR. While not a precise metric, improving MDR based on user feedback will be a crucial measure of quality. At the end of each beta users are supplied a survey via email. These surveys provide more precise scales on the user's satisfaction, perceived usefulness of MDR, which features they found most helpful, what they think can be improved, how valuable they found MDR, and more. These surveys will be tallied at the end to get a score out of ten of how satisfied the user's were, along with their less precise but highly valuable text based sentiments.

5.2 Recruitment Status

Empirical analysis provides the most solid evidence that MDR provides benefits to users in general. Setting up empirical analyses means getting a sample size capable of providing statistically significant data. It is true that effect size can influence this, but for sample groups of size 50 the effect size has to be roughly an 80% increase in order to achieve 95% confidence [4]. As a one member team, recruiting enough people to use the software for several weeks is a significant challenge. In person recruitment via Cal Poly classrooms as well as wearing a sign on my back yielded roughly 100 signups. Online recruitment via A/B tested landing pages and social media got over 400 more signups in a month and a half. Getting people to sign up via email has several technical issues that caused this to be insufficient. If every person who signed up were to download and complete the beta test, this would be sufficient. However, 10% of emails were found to be useless due to people writing their emails incorrectly. This reduced sign-ups to

458, which is still sufficient. The next issue is email engagement rates. Sending out emails with details on the beta proved a huge bottleneck as only 15% of people open and read the emails. Clearly more emails or a more effective approach will be necessary to cultivate sufficient engagement rates. There are further bottlenecks currently as while the majority of personal computer users use Windows, a significant portion of them have mentioned that their main computer runs the OSX operating system. While future versions of the software will be released for OSX, that has not been developed yet and therefore excludes some potential users at this time.

5.3 Personal Assessment

Currently the best measure of assessing the software's usefulness comes from personal use. With over two weeks of constant usage, there is only one error type logged. This error gracefully degrades, since it is the controller not handling chrome developer tools recording properly. This does cause this window to be recorded as chrome and causes more cpu usage due to the recording process. The bug has been filed and will be fixed in the next sprint.

CPU usage has been a large concern that personal data has alleviated. Profiling MDR shows the analytics engine using .1% and the controller using .3% of an I5-4200U CPU on average. These levels are acceptable and should cause no issues for users given that this is among the weakest CPUs available in power saving mode using insignificant amounts of CPU.

From a subjective standpoint tracking time used has been a helpful metric to have automatically collected. Because the only activities classified at this point have been those personally classified, more time is spent doing so. This has not

been a particularly onerous task, with roughly an hour spent on it. The time spent decreases as more and more activities and their classifications are stored for future analysis.

SmartOpen has proven to be remarkably useful beyond its initial use case for automatically opening email. Other uses include opening a habit tracking website before bed, automatically checking google voice, automatically opening anki to help with learning, and even opening linkedin.com once a week. As more uses for this feature are discovered its utility can only grow.

5.4 Full Scale Beta Protocol

As soon as a sufficient sample size of users has been collected, a protocol has been developed for handling the beta testing process. All users will be sent an email and asked to download an installer of the beta software and email if the installation does not show a confirmation page. After they run the installer they should be shown the confirmation page and told to continue using their computer as normal until they receive a notification that testing has begun. [1]

Testing will start 1 week after users start installing the software, providing time for everyone to recieve the email and run the installer. There is a secondary purpose to this week, because this provides a sample of data where the users have not interacted with the software. The timing of this week will prove essential, because holidays or other events could significantly skew that week versus the rest of the experiment. With this concern in mind, the data still will have the potential for uncovering valuable behavioural patterns on a larger sample size.

[1] The download page will also provide informed consent about what data will be collected from the user and the general nature of the experiment.

After the week, the users will be assigned into experimental groups via random assignment. The distribution pattern of users will be weighted to 12.5% active control group, 12.5% passive control group, 25% SmartOpen, 25% block distractions, 25% mindfulness.

5.4.1 Passive control group

These users will not receive any notification when the experiment begins, their data will simply be collected to get a baseline for time usage. After the collection period ends they will be given access to the UI so they can view their data. This is also so they can classify any activities they disagree on being productive or distracting.

5.4.2 Active Control Group

Users in this group will receive a notification granting them access to view their activity data, but will not give them feedback based on that data.

5.4.3 SmartOpen Group

Users in this group will be able to view their activities in addition to the SmartOpen feature. The settings page for SmartOpen will be shown to the user where they can see a description of the feature and a simple one minute example video of how they can set it up with their webmail or email application of their choice. SmartOpen will then open the applications and websites on their schedule automatically. This should increase focus and promote more consistent usage of the applications and websites they configure. Another metric of success for this

group will be how many items they configure, as more items should increase their focus and productivity. The reason specifically that focus should increase is that the items do not open if the user is doing a productive task, but wait for a time in the opening window where the user is on an unproductive task they can be interrupted from. The user will also be suggested to disable email notifications and rely on their configured opening schedule, and at the end they will be surveyed on whether or not they did they did this to gauge the effect of the suggestion.

5.4.4 Block Distractions Group

This group can view how they spend their time, but also are notified after spending 30 minutes on distracting tasks, and have all distracting web pages blocked after an hour of distracting activities. The users will be able to configure how long these time periods are, along with which distractions get blocked and which do not. The success of this method will be gauged by not only if distracting time is reduced, but if productive time increases. User feedback about whether or not this feature works for them will be a critical metric due to the potential for user frustration with blocked distractions.

5.4.5 Mindfulness Group

This group can view how they spend their time, but also will be given a mindfulness exercise each day when the experiment starts. The way it will be administered is that MDR will automatically pop up a notification to do the mindfulness exercise after the user spends 15 minutes on distracting tasks that day. The notification will pop up every 15 minutes up to two times, but the

duration before notification as well as number of notifications will both be configurable by the users. Whether the user follows through on the notification will be recorded, along with the completion rate of the exercise. The exercise itself is based on mindfulness exercises with research proven benefits [10].

5.4.6 Post Experiment Debrief

After the two weeks of the experiment are up, all users will be given a debrief on what part they took in the study and asked to fill out a survey to rate their experience along with any retrospective feedback in textual form.

5.5 Future Feedback

The features and infrastructure developed so far have been fairly low risk to develop with slower feedback cycles from users. Future features will get more specific, costly, and need to be targeted better towards customers that are confident they will pay for the value the new features provide. Accelerating the feedback process is therefore highly valuable. Fortunately the architecture of the software makes it possible to get feedback from users on potential features with less development than prototyping the entire feature. Because all features interact with the user through commands, which have a defined and standard interface the process is simplified. The process starts by developing a protocol for how a feature works. Most features require analyzing user data and translating it into commands. The English description of the protocol for a potential feature can be written out. As long as all commands in the protocol are implemented in the controller, the process can continue. Interested users can be invited to come

work in the same room as the experimenter working with the protocol. A component will be developed so that the user can give the experimenter permission to connect and push commands to their copy of MDR. As the user works in the room the experimenter will follow the protocol and push commands to the user's machine and get feedback from the user on the potential usefulness of the feature. Because agile development works best with a tight feedback loop with users, this tool can increase the advantages of using an agile development process.

Chapter 6

Conclusion

Several major factors are taken into consideration when reaching a conclusion in assessing MDR. The technical achievement is of central importance, but is tightly coupled with human involvement in the project. It was a known factor that the scope of this project was ambitious and far larger than a senior project would normally be.

6.1 The Future

Practical constraints have limited the scope of this project for now. With a renewed effort, more users will be recruited to validate and test MDR. A senior project requires technical accomplishment, diligent execution, and effective communication. With a functioning version of the software, technical accomplishment has been achieved. Diligent execution has definitely been a part of the process. For example MDR itself recorded over 10 hours of productive work on the Saturday of Thanksgiving break. As far as effective communication, this document stands as documentation of what has been achieved so far. While only some parts

of the project have been achieved, the scope of what is done now already fits all three criteria of a senior project. Because of the reflective and academic value of the writing process, this document will be updated in the future with additional milestones and analysis as MDR is deployed to groups of users.

Bibliography

[1] Joshua Brustein. Americans now spend more time on facebook than they do on their pets. http://www.bloomberg.com/bw/articles/2014-07-23/heres-how-much-time-people-spend-on-facebook-daily, 2015.

[2] Jill Duffy. Slife (premium) review. http://www.pcmag.com/article2/0,2817,2405000,00.asp, 2015.

[3] IBISWorld. Operating systems and productivity software publishing in the us: Market research report. http://www.ibisworld.com/industry/default.aspx?indid=1985, 2015.

[4] KissMetrics. A/b significance test. http://getdatadriven.com/ab-significance-test, 2015.

[5] Marion Pick Markus Bhner, Cornelius J. Knig and Stefan Krumm. Working memory dimensions as differential predictors of the speed and error aspect of multitasking performance. http://www.tandfonline.com/doi/abs/10.1207/s15327043hup1903_4, 2009.

[6] Romeo Vitelli Ph.D. Exploring facebook depression. https://www.psychologytoday.com/blog/media-spotlight/201505/exploring-facebook-depression, 2015.

[7] Inc. RescueTime. Find your ideal work life balance. https://www.rescuetime.com/, urldate = 2015-12-09, 2015.

[8] LLC Slife Labs. Slife makes it impossibly easy to keep track of time and stay productive. http://www.slifeweb.com/, urldate = 2015-12-09, 2015.

[9] Techempower. Techempower framework benchmarks. https://www.techempower.com/benchmarks/, 2015.

[10] Diana Winston. Ucla mindful awareness research center - free guided meditations. http://marc.ucla.edu/body.cfm?id=22, 2015.