MOBILE COMPUTING ALGORITHMS AND SYSTEMS FOR USER-AWARE OPTIMIZATION OF ENTERPRISE APPLICATIONS

A Dissertation
Presented to
The Academic Faculty

By

Uma Parthavi Moravapalle

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Electrical and Computer Engineering

Georgia Institute of Technology

May 2019

MOBILE COMPUTING ALGORITHMS AND SYSTEMS FOR USER-AWARE OPTIMIZATION OF ENTERPRISE APPLICATIONS

Approved by:

Prof. Raghupathy Sivakumar, Advisor
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Faramarz Fekri
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Umakishore Ramachandran
College of Computing
Georgia Institute of Technology

Prof. Douglas Blough
School of Electrical and Computer Engineering
Georgia Institute of Technology

Prof. Karthik Ramachandran
Scheller College of Business
Georgia Institute of Technology

Dr. Shruti Sanadhya
Connectivity
Facebook

Date Approved: March 7^{th} 2019

All the power is within you. You can do anything.

Swami Vivekananda

To Amma and Nanna ...

ACKNOWLEDGEMENTS

I would like to express my appreciation to all the people who directly and indirectly helped me over the course of my PhD.

I want to start this list by thanking my advisor Prof. Raghupathy Sivakumar. Siva was an amazing mentor and a great role model to me. Without his guidance, this dissertation would not have been possible. His breadth of knowledge and the clarity on research that comes with this will always inspire me. I learnt to emphasize practicality and relevance of solutions from him. He encouraged me to believe in myself and instilled confidence when I was unsure. Most importantly, he is a kind and patient person who recognized my strengths and weaknesses and created the bandwidth for me to grow. I am extremely lucky to have Siva as an advisor.

I want to express my gratitude to Shruti Sanadhya for her mentorship. She was always available for advice when I needed it. I also want to thank my other labmates - Chao-fang Shih, Bhuvana Krishnaswamy, Yubing Jian, Mohit Agarwal, and Shruti Lall for their support over the years with their encouragement, brainstorming, feedback and friendship. Lonely lunchtimes and boring weekends were rarely an issue.

Even the tiniest success I have seen would not have been possible if not for my family's whole-hearted and unwavering support. I would like to thank my father (Mohan Reddy), my mother (Rajani), my brother (Nanda Kishore), my sister-in-law (Harshita) and my husband (Anirrudh) for their constant prayers and encouragement.

I would like to thank my committee members - Prof. Umakishore Ramachandran, Prof. Faramarz Fekri, Prof. Karthik Ramachandran, Prof. Dough Blough, and Dr. Shruti Sanadhya for their valuable comments and feedback in shaping this dissertation.

Finally, I would like to thank the lord almighty for making it possible.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

SUMMARY

The adoption of mobile devices, particularly smartphones, has grown steadily over the last decade, also permeating the enterprise sector. Enterprises are investing heavily in mobilization to improve employee productivity and perform business workflows, including smartphones and tablets. Enterprise mobility is expected to be more than a $250 billion market in 2019. Strategies to achieve mobilization range from building native apps, using mobile enterprise application platforms (MEAPS), developing with a mobile backend as a service (mBaaS), relying on application virtualization, and employing application refactoring.

Enterprises are not yet experiencing the many benefits of mobilization, even though there is great promise. Email and browsing are used heavily, but the practical adoption of enterprise mobility to deliver value beyond these applications is in its infancy and faces barriers. Enterprises deploy few business workflows (<5 percent). Barriers include the heavy task burden in executing workflows on mobile devices, the irrelevance of available mobile features, non-availability of necessary business functions, the high cost of network access, increased security risks associated with smartphones, and increased complexity of mobile application development.

This dissertation identifies key barriers to user productivity on smartphones and investigates user-aware solutions that leverage redundancies in user behavior to reduce burden, focusing on the following mobility aspects:

(1) Workflow Mobilization: For an employee to successfully perform workflows on a smartphone, a mobile app must be available, and the specific workflow must survive the defeaturization process necessary for mobilization. While typical mobilization strategies offer mobile access to a few heavily-used features, there is a long-tail problem for enterprise application mobilization, in that many application features are left unsupported or are too difficult to access. We propose a do-it-yourself (DIY) platform, Taskr, that allows users

at all skill levels to mobilize workflows. Taskr uses remote computing with application refactoring to achieve code-less mobilization of enterprise web applications. It allows for flexible mobile delivery so that users can execute spot tasks through Twitter, email, or a native mobile app. Taskr prototypes from 15 enterprise applications reduce the number of user actions performing workflows by 40 percent compared to the desktop;

(2) Content sharing (enterprise email): An enterprise employee spends an inordinate amount of time on email responding to queries and sharing information with co-workers. This problem is further aggravated on smartphones due to smaller screen real estate. We consider automated information suggestions to ease the burden of reply construction on smartphones. The premise is that a significant portion of the information content in a reply is likely present in prior emails. We first motivate this premise by analyzing both public and private email datasets. We then present Dejavu, a system that relies on inverse document frequency (IDF) and keyword matching to provide relevant suggestions for responses. Evaluation of Dejavu over email datasets shows a 22 percent reduction in the users typing burden;

(3) Collaboration: Even though many business processes within enterprises require employees to work as a team and collaborate, few mobile apps allow two employees to work on an object from two separate devices simultaneously. We present Peek, a mobile-to-mobile remote computing protocol for collaboration that lets users remotely interact with an application in a responsive manner. Unlike traditional desktop remote computing protocols, Peek provides multi-touch support for ease of operation and a flexible frame compression scheme that accounts for the resource constraints of a smartphone. An Android prototype of Peek shows a 62 percent reduction in time to perform touchscreen actions.

CHAPTER 1

INTRODUCTION

Enterprises continually focus on boosting employee productivity. While they approach this using several strategies, mobilization is seen as a game changer. Mobilization lets an employee rely on mobile devices to perform business functions while away from the desktop. This is inspired by two contemporary trends increasing reliance of enterprises on software applications for essential business functions and the rising adoption of mobile devices, in particular smartphones, among enterprise employees. Several studies show that employees gain up to 81 minutes of work and personal time through mobilization [1–3]. Enterprise mobility is expected to be more than a $250 Billion market [4].

While the earliest versions of enterprise software were purely in-house ledger applications that supported a few business functions, enterprise software today is deeply integrated within the day-to-day operations of an enterprise. A modern enterprise relies on several software applications for essential business functions, such as customer relationship management [5–8], human resource management [9–12], enterprise resource planning [13–16], business intelligence [17–20], content management [21–23], communication [24–26], accounting [27, 28], enterprise asset management [29–31], supply chain management [32–34], and product lifecycle management [35, 36], among others. Enterprise spending on software applications is expected to reach $435 Billion by 2019 [37].

In tandem with the rising dependence on software for business functions is the evolution of enterprise software applications within the enterprise. Smartphone growth has been explosive during the last decade. By 2022, 81 percent of the United States population is expected to own a smartphone [38]. Even traditionally conservative enterprise sectors are adopting mobile devices at a blistering pace, driven by a clear return-on-investment in the form of higher employee productivity, reduced paperwork, and increased revenue. With a

significant amount of business functions now occurring over enterprise software, the focus of the enterprises to enable mobilization is inevitable. This can be attributed to the many advantages of mobility-accessibility, convenience, and context-awareness. The ubiquity of smartphones implies that employees now have access to a computing device for most of the day [1]. Therefore, if all business functions are made possible on a smartphone, the benefits are clear critical business functions can be addressed faster, and work can be conveniently be conducted from different locations. An added advantage to using smartphones for business is the creation of a new generation of context-aware enterprise apps. All these drivers of enterprise mobility directly translate into employee productivity benefits. Today, 98 percent of enterprises allow their employees to use mobile apps for work either on enterprise provisioned devices or on personal smartphones [40].

Despite the great promise of mobilization, many of its benefits are not fully realized. The average global 2000 enterprise uses 1031 software applications for business processes [40]. This includes on-premises applications such as SAP or Oracle; cloud-based applications such as Salesforce and Workday; and homegrown applications purpose-built using web, .NET, Java, and even legacy green screen systems. However, 70 percent of enterprises provide fewer than 10 mobile apps to their employees [41]. Also, more than a third of 500+ enterprises surveyed in [42] provide access only to basic mobile apps such as Email, Web browser or Calendar on their employees' smartphones. If these enterprises were to mobilize several other applications, 15 percent productivity gains are expected [42]. *This dissertation's primary focus is to investigate the reasons for the unrealized potential of enterprise mobility and propose solutions to overcome them.*

In the rest of this chapter, we contend that creating mobile apps from complex desktop applications results in a long-tail problem, where the business functions for each individual user are either not supported by the app or are difficult to accomplish. We then argue that user-aware strategies to mobilize applications are necessary to overcome the *long-*

[1] A recent study shows that most users have access to their smartphones nearly 21 hours a day. In contrast, they only have access to their Desktop for 8 hours in a day [39].

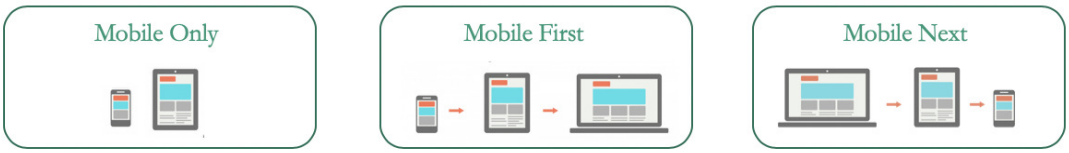


Figure 1.1: Enterprise Mobility Strategies

tail problem. Finally, we discuss our research focus and the specific contributions of this dissertation toward three different aspects of enterprise mobility: content creation over email, workflow mobilization, and collaboration on mobile apps.

1.1 Enterprise Application Mobilization

Application mobilization is the process of creating mobile apps and services that enable users to perform their work tasks on a smartphone. Strategies to mobilize business functions fall within three categories (see Figure 1.1: (i) A *Mobile-Next* strategy, where the business functions within existing desktop applications are ported into a smartphone app. Applications such as Salesforce, SAP, Peoplesoft, and Sharepoint were initially created for the desktop and later ported to a variety of smartphone platforms; (ii) A *Mobile-First* strategy, where a smartphone app and backend services (that support both smartphone and desktop platforms) for the business functions are initially created and subsequently, extended to the desktop platforms. Evernote, Databox, and Rapid Value are some examples of applications for which the smartphone app was created first and later extended to the desktop; and, (iii) A *Mobile-Only* strategy, where only a smartphone app is created for the business functions. IBM Dynamicbuy, APTTUS, Domo, and Workflow are some examples of smartphone-only enterprise apps.

With the mobile-first and mobile-only strategies, a smartphone is the primary device with which a user is assumed to execute business functions. Therefore, smartphone apps and backend services are created ground-up and optimized for a smartphone user. On the other hand, with a mobile-next strategy, enterprises transform existing desktop applications

that are already deeply integrated within their operations into smartphone apps. This strategy seeks to leverage the user's familiarity with performing the business functions on the desktop application. Furthermore, as the enterprises can reuse existing infrastructure resources (codebases and services), the cost of mobilization efforts and time to production are relatively low. Therefore, among the three strategies, the mobile-next strategy is prevalent among enterprises. In this dissertation, we restrict our focus to application mobilization using a mobile-next strategy.

Enterprise applications are complex in nature and contain thousands of features to support business functions across several functional roles within the enterprise. *A single task within an enterprise application often requires involvement from users across several different functional roles.* For example, consider employee business trip management within an HR application (like Oracle Peoplesoft). A trip to a client location to close a sales deal involves the following steps: (i) The employee submits a travel request providing a business justification and preferences; (ii) An administrative assistant processes this request, prepares a tentative itinerary and submits it for approval; (iii) An accounting manager cross-checks this itinerary for funding and requests supervisor approval; (iv) The employee's manager checks the business justification and provides authorization; (v) The administrative assistant hands this itinerary to a travel agent for booking; (vi) The employee completes the trip and sends a few receipts to the administrative assistant for approval; (vii) The assistant prepares a travel reimbursement form with the receipts; (viii) The accounting manager approves this form and instructs the payroll department to reimburse the funds; and, finally, (ix) The payroll department releases the funds to the employee's bank account.To support a simple use-case of employee business travel, an HR management application needs to support the different workflows involved in trip management for all functional roles involved in this process: employee, administrative assistant, accounting manager, employee manager, travel agent, and payroll. Given the feature-packed nature of enterprise applications, workflows are often complicated to execute and require the user to perform multiple actions.

Some of these applications have undergone several development cycles over time with new features added within each cycle[2]. Moreover, the workflows within these applications are primarily designed for the stationary user who views the application on a desktop with a large screen.

To enable mobilization, the same complex workflows now have to be ported to the smartphone. This is a non-trivial challenge because unlike desktops, smartphones are resource constrained. They have less on-screen real estate, fewer computation capabilities, access unreliable wireless networks with lower speeds, rely on batteries with limited power, and have low on-device storage capacities. Furthermore, smartphone users have shorter attention spans and higher usability requirements [43]. With these constraints, a single mobile app is likely to support only a subset of the features available in a desktop application. These features have to be carefully chosen to maximize usability. We call this *the defeaturization* of enterprise applications.

1.1.1 User-Aware enterprise mobility

Defeaturization must occur for successful mobilization. Defeaturization is done either by the enterprise or the application vendor based on the needs of an average user for the application. The result of defeaturization is a smartphone app which contains a carefully curated subset of features from the original application. To serve a large user base, this subset is tailored to include a few heavily-used features from the original desktop application. Recall that enterprise applications contain features serving various functional roles in the enterprise. If the complex enterprise application is defeaturized into only one mobile app, it is highly unlikely that this mobile app is optimized for all of the functional roles using the application. Therefore, the inherent nature of the choice of features in the defeaturization step excludes certain users from truly adopting mobile solutions for their work. This generalized defeaturization approach results in a long-tail problem of application mobilization.

[2]The first version of Peoplesoft was released in 1989

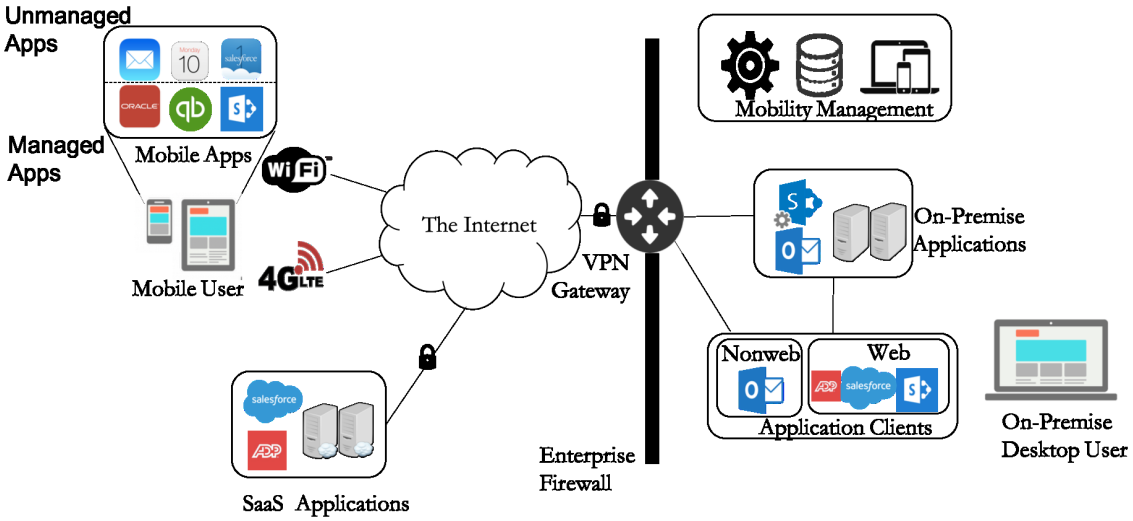


Figure 1.2: Enterprise Mobility Architecture

If the smartphone apps are built for the average user (or one functional role), it is unavoidable that there are some users whose workflows are either not supported in the smartphone app or are very burdensome to perform.

The long-tail problem can indeed be avoided by not defeaturizing the application at all. However, given the resource-constrained nature of smartphones, including every feature from the desktop application in the smartphone app would either not be possible or render the app unusable. To solve the long-tail problem, the defeaturization has to be tailored to the needs of each user. A trivial solution to this problem is for the enterprise to make several mobile apps (customized for each user or each functional role). However, given that a single mobile app costs roughly $250,000 to develop [44], it is an expensive proposition for the enterprise to make several apps. This calls for an alternate approach to mobilization where defeaturization is performed efficiently and in a user-aware manner.

1.1.2 Enterprise Mobility Architecture and Mobilization Challenges

Figure 1.2 shows a simplified version of a typical enterprise mobility architecture. Enterprise applications follow a client-server architecture with application servers hosted either on-premise (within the secure enterprise network) or on the cloud connected to the en-

terprise network through a secure VPN gateway. These applications can be used by an on-premise user with application clients that are either stand-alone software applications or through a browser (web application clients). Off-premises users can also access the applications through the secure VPN gateway. The enterprise supports mobile users who may be off-premises by issuing them in house custom-made or vendor developed clients in the form of mobile apps. The apps access the enterprise application backend servers and data through either WiFi or cellular networks. For security and data protection purposes, the enterprises require the apps to access enterprise data through secure VPN. For managing and monitoring mobile clients, enterprises deploy mobility management suites like VMWare Airwatch or Mobile Iron. With such mobility management solutions, enterprises can keep track of which mobile devices are active, what data is being accessed, and configure policies suitable for these categories.

In the context of the enterprise mobility architecture shown in Figure 1.2, user-aware application mobilization results in several challenges, some of which are outlined below.

- *Complexity:* Enterprise desktop applications have complex workflows involving several user actions to accomplish a goal. These complex workflows, if implemented as-is within a smartphone app so that the users expect to complete workflows within a few taps, can result in poor usability.

- *Availability*[3]*:* The defeaturization process involves choosing a subset of workflows from the desktop application to include in the smartphone app. This decision is made either by the enterprise or the application vendor, based on the perceived needs of the applications user base. This, however, means that some features will not be included in the smartphone app.

- *Network:* Desktops are typically connected to wired networks that offer better browsing speeds and are reliable, compared to wireless networks used by smart-

[3]Note that this definition of availability is different from the more traditional meaning associated with availability in the context of distributed systems.

phones. Furthermore, the cost of wireless access is much higher ($10/1GB, cellular) than wired access ($0.16/1GB). If the workflows originally designed on the desktop with for high network speeds are performed on the smartphone, the user may end up with poor app response times and expensive data usage bills

- *Compatibility:* Desktops use different platforms and operating systems than smartphones. When a desktop application is ported to a smartphone, some workflows may break if they use APIs that are not yet available or incompatible on the smartphone platform.

- *Computation:* Despite the advances in smartphone processors, they are still not as fast as their desktop counterparts. If the workflows that need heavy computation power (e.g., sorting, indexing, etc.) are to be preserved in the smartphone, usability issues may arise (e.g., application hanging or an unresponsive screen).

- *Security:* As enterprises now allow users to use personal smartphones for work, they must ensure that the mobile apps and data are secured when accessed outside the enterprise networks. A recent survey of IT decision-makers revealed that more than half of enterprises are concerned about data leakage, unauthorized access to company data/systems, and downloading unsafe apps, content, and malware. With these security fears, enterprises may decide against developing applications.

1.2 Research Focus

In this dissertation, we restrict our focus on the first two of the outlined challenges, i.e., complexity and availability. In this context, we investigate the following three different aspects of mobility: (i) Content creation over email on smartphones, (ii) Workflow mobilization, and (iii) Collaboration with smartphone apps. For a user intending to perform specific tasks within these aspects (on a smartphone), we argue that existing smartphone apps either do not support all the tasks the user intends to perform and/or the burden of

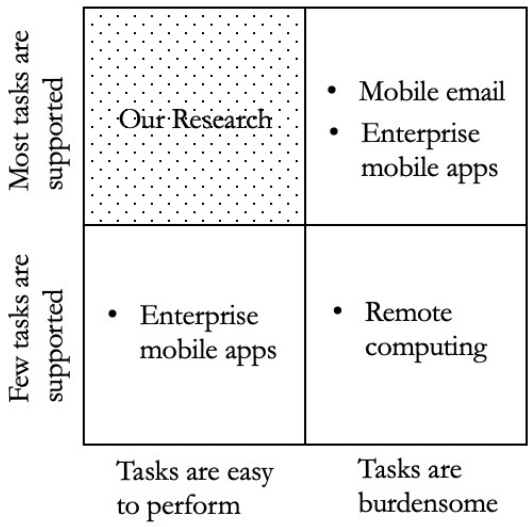


Figure 1.3: Research Focus Landscape

performing the tasks supported on the smartphone is high. We then propose user-aware strategies and solutions to either reduce the complexity of performing tasks or increase the availability of tasks on the smartphone.

1. Content Creation with Email

Email is a dominant form of information sharing within an enterprise. An enterprise employee spends an inordinate amount of time reading and responding to email. This email overload results in the average enterprise employee spending 28 percent of their workweek on email [45]. This is further aggravated because a large portion (up to 70 percent) of these emails are opened on smartphones. Given that 30 percent of email replies are more than 100 words, it is burdensome to respond on a smartphone.

In this context, we consider automated information suggestions that can make replying easier. A significant portion of the information content of a reply is likely present in prior emails. We first motivate this premise by analyzing different publicly available email datasets and showing that nearly 60 percent of email responses are very similar to the information contained within the past emails of that user. We then present *Dejavu*, a system that

leverages the previous emails within the users own mailbox to provide relevant suggestions for responses. We show that *Dejavu* s simple keyword and Inverse Document Frequency (IDF) approach can deliver effective suggestions for 1 in 3 responses. We propose several optimizations to *Dejavu*, known as *Dejavu++*, to improve the efficiency of computing suggestions and increase the relevancy of suggested responses.

On the landscape shown in Figure 1.3, the task of typing email responses on a smartphone is supported but is burdensome. *Dejavu* seeks to reduce the burden by offering automated suggestions.

2a. Workflow Mobilization

An average global 2000 company uses more than 1000 applications for its business processes. Fewer than one percent of these are made available on a smartphone. For an enterprise employee to successfully perform workflows on a smartphone, not only does a mobile app need to be available, but the specific workflow must be available despite the defeaturization process necessary for mobilization. While typical mobilization strategies offer mobile access to a few heavily-used features, there is a long-tail problem for enterprise application mobilization. A large swathe of application features are either not supported or are too difficult to access from the mobile.

In this context, we propose a do-it-yourself (DIY) framework, *Taskr*, which allows users, irrespective of skill level, to mobilize the workflows themselves, without requiring any support from either the application vendor or the enterprise. For such a solution, we identify a category of workflows, called Spot Tasks within web-based enterprise applications, that are suited for robust DIY mobilization. These tasks are simple workflows that allow users to interact with the desktop application, but only on a single page. *Taskr* collects specifications of the workflow to be mobilized by observing the users workflows. *Taskr* uses remote computing with application refactoring to allow codeless mobilization of enterprise applications. *Taskr* then delivers these workflows to the user not only through a

custom client app, but also through Twitter, email, and texting. The *Taskr* prototype reduces user burden by 40 percent when performing workflows from nine enterprise applications.

In the context of Figure 1.3, for the long-tail users, existing mobile apps either do not support the workflows at all or have workflows that are burdensome to execute. *Taskr* empowers users to create tailored mobile apps, thereby both increasing the number of workflows on a smartphone and reducing the complexity of executing the workflows.

2b. Application APIfication for workflow mobilization

In application refactoring, the application is hosted as-is on the enterprise cloud. The user interacts via a smartphone-optimized UI. These services require applications to provide APIs to map the smartphone UI to the web application UI accurately. Of the strategies available to APIfy applications, a front-end only approach, based on intelligent screen scraping, is particularly attractive as it can APIfy a host of applications without support from the applications themselves. Front-end strategies, however, must accurately and reliably identify UI elements within the application. We show that simple approaches, which depend on graphical coordinates or the position of an element with respect to a fixed anchor in the application layout, are not robust enough for APIfication. Thus, we present *Trackr*, an algorithm that uses quorum fingerprinting to track elements. We discuss several optimizations to this baseline version. By analyzing changes to real-world web-applications, *Trackr* improves the tracking accuracy of UI elements within enterprise applications by 55 percent compared to a related approach. We also demonstrate use of *Trackr* through a dashboard smartphone app that monitors values throughout different websites within one mobile app.

3. Collaboration:

Business processes within enterprises often require employees to collaborate as a team. Few enterprise mobile apps are designed for collaboration, allowing two employees to work

on an object from two unique devices simultaneously. We argue that remote computing can allow for collaboration even when the application itself doesnt support it. While there are several desktop remote computing solutions available, they cannot be applied as-is for mobile-to-mobile remote computing. In this context, we present *Peek*, a mobile-to-mobile remote computing protocol for smartphones that allows users to interact with an application remotely in a responsive manner with (i) multi-touch support, (ii) context association, and (iii) multi-modal frame compression. An Android prototype of *Peek* shows a 62 percent time reduction to perform some common touchscreen actions.

In the context of Figure 1.1, existing remote computing protocols, when applied to smartphones, do not support many touch-screen gestures and are prohibitive due to high resource costs. *Peek*'s multi-touch capability with resource efficient frame compression aims to reduce the complexity of interaction and increases the availability of supported touch-screen gestures.

1.3 Thesis statement

Complex enterprise applications when adapted for the resource-constrained mobile devices result in complexity and availability issues. These issues can be effectively addressed by user-aware strategies that leverage redundancies in user behavior.

1.4 Thesis Organization

This dissertation is organized as follows: In Chapter 2, we discuss work in the commercial and research domains related to the problem statements. In Chapter 3, we discuss details of our work on a user-aware automated reply suggestion, *Dejavu*. In Chapter 4, we present the user-aware application mobilization platform, *Taskr*. In Chapter 5, we introduce *Trackr*, a front-end APIfication approach to aid application mobilization services. In Chapter 6, we discuss details of *Peek*, a mobile-to-mobile remote computing protocol. In Chapter 7, we discuss how the four individual solutions can be integrated within enterprise mobility

architectures. Finally, in Chapter 8, we outline additional research directions, concluding our arguments in Chapter 9.

CHAPTER 2

LITERATURE SURVEY

In this chapter we discuss commercial products and research works related to the three mobilization aspects we consider in this proposal.

2.1 Content Sharing

2.1.1 Commercial Solutions:

Enterprise worker is typically exposed to several knowledge sources during his work day. Information relating to his work is distributed in all these sources. Due to the diverse nature of types of information (documents, emails, IM etc.) and the applications (Dropbox, Gmail, local storage, Slack etc.), it is hard for the typical enterprise worker to quickly retrieve relevant information. Companies like Coveo, Sinequa, Attivio, etc., provide an unified index on these various sources of information and thereby helping the user retrieve relevant information in a timely fashion. However, these solutions do not specifically focus on automatically helping users construct email responses.

2.1.2 Research Solutions:

Email optimizations

The problem of information overload in email was first recognized in [46] in 1996. Several solutions have since been proposed to optimize email to combat email overload. [47, 48] suggest intelligent categorizing techniques to manage information efficiently. Few works used content summarization techniques to extract summaries from email [49–51]. These summaries could then be used for better presentation of email lists. [52–54] identify certain speech acts in email such as - statement, request, propose (meeting), amend, commit,

deliver .etc, to better help the user track the status of an ongoing task. Few other solutions prioritize each email as being important or not to help user quickly deal with and respond to a large inbox [55, 56] Just like the semantic web, semantic email has been proposed by [57] where in each email is tagged with certain semantic information that can be leveraged at a later stage for context specific applications. The semantic information can then be used to easily achieve tasks such as event planning, information dissemination, report generation, auction/giveaway, etc. Apart from these solutions, all email clients provide a search feature to retrieve relevant information easily in an overloaded inbox. There are about a 900+ startups working on optimizing various aspects of email and providing new features for increasing productivity.

Reply Prediction

[51] explored the idea of predicting whether the email needs a reply or needs an attachment. However, they do not predict the content of the reply or which attachment to include. Some works [58, 59] have explored identifying experts through email conversations. This information is very useful and can be used to direct the conversation on a topic towards the expert and elicit responses. However, these works do not provide a way to lookup information that is potentially available in the user's own inbox and construct responses from there.

Question-answering systems are another class of research works that can be adapted to provide automated response suggestions. Of particular interest are the systems trained to answer natural language questions using the unstructured information within the corpus. These systems can be broadly classified into two categories [60]- (i) Information Retrieval based systems[61–66], and (ii) Knowledge based systems[67, 68]. Given a question, information retrieval systems find portions of text within the large corpus that may contain the answer to the question. On the other hand, knowledge-based systems process the corpus to extract facts and then allow queries on these facts. Most of these question answering

systems only consider factoid questions, where the answers are expected to be short and expressed using a word or a phrase. Enterprise email queries are mostly non-factoid requiring long informational responses. For example, the Avocado IT enterprise email dataset has an average response length of four sentences.

Some retrieval systems can be adapted for automated response suggestions. Watson Discovery [62] is one such popular commercial information retrieval system. Given a collection of documents, Watson Discovery can extracts information from these documents such as sentiment, named entities, concepts, semantic roles, etc. A user can retrieve passages or whole documents using queries on the extracted information in natural language or in a proprietary query language. While discovery works well for documents such as news articles, Wikipedia pages, etc., it is not optimized for a conversational corpus like email.

The closest related works to automated response suggestions for email are Smart Reply [69], Quick Type [70] and Outlook's suggestions [71]. These systems encode an Inbox email through a recurrent neural network and extract context. This context is then used to predict either coherent responses from another recurrent neural network [69, 71] or next probable word in the current response [70]. The recurrent neural networks are trained on the user's inbox. However, these solutions only construct generic non-informational phrases.

2.2 Workflow Mobilization

2.2.1 Commercial Solutions

(i) *Custom homegrown solutions:* Some enterprises develop custom native apps for a target smartphone platform. Eg., SupportCentral from General Electric; The limitations with this approach include rewriting of code for all (or partial) functionality for the application, and separate development effort for different smartphone platforms and hence can be prohibitive in terms of developer time/effort required and cost of app development. (ii) *Vendor applications:* The enterprise can use existing vendor native apps available on the app market if one is available. However, the functionality available in the app cannot be controlled

by the enterprise and most applications available in the app markets have a very reduced functionality when compared to their desktop counterparts. Eg., Salesforce1,Oracle BI, SAP Fiori Client, etc; (iii) *Mobile Backend as a Service (mBaaS):* Enterprises can build rich mobile applications by using mobile-specific backend features available as libraries. These features include authentication services, data storage, file storage, integration with third-party cloud services, analytics etc. and can be used to develop applications using any target platform SDK of choice. Eg: AnyPresence [72], etc; However, these solutions still require manual development effort from the enterprises. Also, once the mobile app is built, it remains the same for all users irrespective of their preference and usage behavior. (iv) *Mobile Enterprise Application Platform (MEAPs):* Enterprises can develop mobile applications using custom application development platforms wherein apps need to be developed once and can be deployed on all target platforms. Since mobilization through MEAPs requires custom development platforms, they require training effort on the part of developers to learn the platform. Further more, the features available in the final product are limited by the features provided by the MEAP platform. Eg:Appcelerator [73], etc.

2.2.2 Research Solutions

Mobilizing workflows for the mobile device has been explored by several research works in the past. PageTailor[74] is a system that transforms web pages that have been designed for the Desktop into smartphone friendly views. The users adapt the webpage by moving, resizing or repositioning the UI elements. This new custom organization is remembered and applied to the webpage and similar ones on subsequent visits. Page Tailor is specific to commercial content based web pages. On the other hand enterprise applications are far more complex. Modeap [75] transforms PC applications to mobile web browser based applications by deconstructing the applications to graphical primitives on the PC end, and reconstructing them on the browser end. However, they do not defeaturize of the complex PC applications, and is specific to PC based applications and not web based applications.

Merlion [76] uses remote computing to allow users to access a Desktop application features from a smartphone. Defeaturization is done by allowing the user to define a subset of UI elements to be visible. However, this requires a complex configuration process by the user. Feedcircuit[77], Highlight [78] and Flashproxy [79] are solutions that allow users to access web applications from mobile phones that lack certain capabilities, such as JavaScript and Flash.

Forms2Dialog [80] is a solution that converts web based forms to speech dialogs, that can be accessible over the smartphone. W3Touch [81] is a webpage instrumentation toolkit that allows developers to find potential problems on their websites when used on the mobile browser (particularly touch screen problems). The developers can later on fix these problems. However, this solution fixes issues on existing pages and does not mobilize web pages. [82] presents a method to display HTML tables with a mobile friendly format. [83] dynamically transforms web pages for mobile browsing and suggests a speech interface for better navigation and usability. [84] proposes a control extraction method that can efficiently extract part of the web pages. Mobitran [85] refactors the original desktop application for mobile user by splitting the page into blocks, rearranging blocks and displaying only the content relevant while preserving javascript and css behaviors. Each of these works are either very specific to a certain class of web pages (forms, tables, etc.) or do not defeaturize making them ineffective to be utilized for mobilizing complex applications in the enterprise scenario.

2.3 Front-end APIfication

The problem of reliably fingerprinting UI elements within a web application has been explored in the past in different contexts. XPath[86] is a widely adopted standard with syntax to describe elements within an XML/DOM tree. Using XPath syntax, a path for traversal within a DOM tree can be specified between two elements. For example, $//html//body[1]$ is the XPath expression to reach *body* by traversing to html's second child. However, XPath

only provides a syntax and it is upto the developer to create a fingerprint with it. Several optimizations[87, 88] have been proposed to interpret XPath. In [89], an element's path from the root of the DOM tree is used as one of its features, but in the context of enhancing mining. [90] uses the shortest path from the nearest ancestor in the DOM tree with an HTML attribute ID as a fingerprint. Here, the context is to record user actions. [91] uses path from the root in conjunction with parent and immediate siblings to identify an element for information extraction. In [92], the authors propose using subtree information for each element in a DOM path. These fingerprints assume a consistent DOM for the web application, which does not hold true in reality. We later show that these single-path based fingerprints do not perform well in dynamic scenarios. [93, 94], use visual features of the page to learn and extract templates for elements. This layout structure can then be leveraged to create fingerprints. However, generating fingerprints based on visual features is not feasible for a majority of secondary services as it not only requires a large amount of annotated training data but also takes a lot of time.

2.4 Collaboration

There are several remote computing protocols for desktops in use today. For example, RDP (Microsoft)[95],RFB (VNC)[96], ICA (Citrix)[97], etc. For mobile thin clients, some optimizations have been proposed in related literature. SmartVNC[98] reduces the burden of doing tasks in a remote computing session from smartphone to a desktop, by identifying macros. Mobidesk[99] proposes WAN traffic optimization for mobile thin clients. Modeap[75] uses translation between graphical primitives of desktop and those of a mobile web browser. [100] and [101] are other solutions that target gaming and multimedia delivery on smartphones, respectively. Yavnc [102] is a VNC based solution that presents a view of the desktop applications on the smartphone. However, all these solutions assume the server is a desktop and are not applicable to a mobile-to-mobile remote computing scenario.

CHAPTER 3

DEJAVU: ASSISTED EMAIL REPLIES FOR REDUCTION OF REPLY BURDEN ON SMARTPHONES

Enterprises today are investing heavily in their mobile workforce with an eye toward boosting productivity and customer service. 91% of mobile workers in enterprises use a smartphone for their work. On the other hand, with information now ubiquitously accessible, job functions of enterprise employees increasingly involve handling, using, or analyzing information. The juxtaposition of these two trends: increasing reliance on access to information, and ubiquitous mobile connectivity — forms the context for this chapter. We specifically focus on one dominant form of information sharing within enterprises – Email. The average enterprise employee sent/received 126 emails per day in 2015 [103]. This deluge of emails results in an average enterprise worker spending 28% of her work time in reading and responding to emails [45]. A large portion (70%) of these emails are opened on a mobile device.

The challenge we explore in this chapter is the burdensome experience of typing replies to emails using the smartphone's small on-screen keyboard. A recent study has indicated that over 30% of all email replies are over 100 words long [104]. Assuming the typing speed of an average user on a smartphone to be 20 words a minute, it takes more than 5 minutes to type a 100-word email response on a smartphone. This directly translates to productivity-related costs for enterprises.

One approach to reducing this burden is to automatically generate suggestions for the content of email replies, which the user can select, modify and send. The content of a typical email response can be classified into two categories: non-informational (e.g., generic words and phrases such as 'okay for a meeting', 'sure', etc.) alternatively, informational (specific responses such as an address, a budget proposal, etc.). There are existing so-

lutions that perform email reply assistance by suggesting appropriate non-informational content (Google's Smart Reply [69], Outlook's suggestions[71] and Apple's Quick Type [70]).

In this chapter, we explore if such assistance is achievable for the *informational content* of the replies. Specifically, we ask the following question: *For a mobile user, if the information required for a reply to an incoming email is available in past emails within the inbox/sent-box/other-folders of that user, could that information be identified, retrieved, and presented to the user in a fashion that eases the burden for the reply construction?* The goal of such an informational email reply suggestion solution is not to replace the existing non-informational suggestion solutions but to compliment them with informational content[1].

In answering the above question, we make the following key contributions: (1) We use publicly available email datasets (Enron Corporation email dataset [105], Avocado IT email dataset [106] and Hillary Clinton email dataset) to analyze the potential for retrieving information from existing emails to help in response construction; In total, we analyze 364,135 emails belonging to 36 different users, and show that the results are quite promising with the percentage of responses that have a 60% similarity match with past emails being 60.41% for three past emails; (2) We demonstrate the feasibility of suggesting informational replies through a simple algorithm *Dejavu* that is based on a keyword match between the email being responded to and past emails. Using a prototype, we show that this simple approach is capable of providing effective suggestions nearly 27.3% of the time; (3) We propose *Dejavu++*, an optimization of *Dejavu*'s keyword matching algorithm. *Dejavu++* reduces the computation complexity of finding suggestions through topic filtering and improves the relevance of suggested replies by utilizing the implicit user feedback available through partially typed responses. (4) Finally, we expand the sources of informational suggestions

[1]It is worth noting that this question is easily extensible to include not just past emails but also other sources of content such as stored files, IM history, online content repositories, the public web, etc., but we restrict the focus of this paper only to past emails.

to other user's email mailboxes and demonstrate that some emails can indeed benefit with suggestions using information outside the user's mailbox.

Through evaluation of *Dejavu* and *Dejavu++*, we show that is tremendous scope in reducing the user burden through informational reply suggestions. Given the recent advances in natural language processing and information retrieval, we hope that this work opens the doors for algorithms that provide more tailored suggestions in the future.

3.1 Motivation

The goal of this section is to establish the potential for effective automated suggestions in assisting email response construction. We do this by analyzing multiple publicly available datasets- ENRON is an email dataset comprising of emails made public during the US SEC investigation of Enron Corportation for fraud; HILLARY is an email dataset made public during the recent investigation into the user of a private email server by former Secretary of State and Senator Hillary Clinton; AVOCADO is an email dataset comprising of mailboxes of employees within a defunct Information Technology company; At a high level, the analysis is performed using a custom-built python tool that for every *reply* in the users Sent folder determines how much the non-trivial keywords and content in that email matches with any prior email(s) in the Inbox, Sent, and other folders of the user [2].

3.1.1 Datasets

Specific details of the three datasets are as follows:

ENRON:

The Enron email corpus [105] is one of the largest available email datasets and consists of 150 email accounts (approximately 500,000 emails) of high-level executives of the Enron corporation. This dataset was made public by the Federal Energy Regulation Commission

[2] We refer to email responses as *replies* in the rest of this chapter in keeping with common usage standards.

Table 3.1: The AVOCADO dataset

User	Dataset ID	#emails	User	Dataset ID	#emails
1	55	33931	11	277	13261
2	173	18592	12	57	12969
3	216	18211	13	8	12957
4	144	16309	14	107	12861
5	196	15824	15	233	10809
6	178	15743	16	80	9517
7	84	14666	17	117	9216
8	63	14455	18	206	9162
9	7	13317	19	245	9155
10	167	13265	20	281	7963

during its investigation into financial irregularities and insider trading allegations. In this chapter, we restrict our focus to a subset of 15 users with a total of 74007 emails. The distribution of emails is presented in Table 3.2[3]. The dataset by default consists of raw email dumps separated into several folders such as Inbox, Sent, Draft, etc., and also any other user-created subfolders. Multiple copies of some emails were stored across different folders. Empirically, we determined that some folders such as _sent_mail, _sent, all_documents, deleted_items, discussion_threads almost always contained duplicate emails and excluded them. We categorize all the email dumps in any folder whose name contains 'sent' as sent emails and all the other emails as received emails[4].

HILLARY:

This dataset contains the publicly released emails belonging to former Secretary of State and Senator Hillary Clinton. The emails were initially released as raw a PDF file with many lines redacted. These emails, belonging to Ms. Clinton's private email server during her tenure as the U.S. secretary of state were released to the public following a controversy, wherein Ms. Clinton was alleged to have violated federal government rules by using her

[3]Note that this subset includes controversial names such as Skilling, former president, and C.O.O. of Enron, who was convicted of federal felony charges for Enrons financial collapse. More details on specific roles of the employees can be found at [107].

[4]Different users used different email clients. The folder structure for each of these users was hence different.

Table 3.2: ENRON dataset

ID	Employee	# emails	ID	Employee	# emails
1	Hayslett, R	2554	9	Sanders, R	7329
2	Arnold, J	4898	10	Neal, S	3268
3	Kitchen, L	5546	11	Lokey, T	1156
4	Farmer, D	13032	12	Steffes, J	3331
5	Kaminski, V	12363	13	Derrick, J	1766
6	Skilling, J	4139	14	Causholli, M	943
7	Maggi, M	1991	15	Geaccone, T	1592

private email server for communication instead of email accounts hosted on federal government servers. These emails have been subsequently cleaned up for analysis and released in the form of a CSV document. This dataset has a total of 7945 emails.

AVOCADO:

This public dataset (released in 2015) contains complete Personal Storage Table (PST) dumps of mailboxes (containing both emails and attachments) from 279 users of a now-defunct information technology company[106]. The name of the company is anonymized and referred to as 'AVOCADO IT' within this dataset. While the original dataset consists of 279 users, we focus on a subset of 20 users (details shown in Table 3.1) with the largest mailbox sizes within the company. This subset of 20 users has 282,183 emails in total.

3.1.2 Processing

The three datasets consist of raw email data and are pre-processed for further analysis. A raw email starts with header data, followed by body content and any attachments. If the email is a reply, the email clients quote the original message (to which this email is a reply) along with the email body. The format of these quotes differs for different clients. Sometimes the quoted text is marked with '<'. In other cases, the quoted text follows lines such as '—Original Message——'. Through heuristic rules made from careful observation of the datasets, we scrub the quoted text from the reply text. At this stage, we also add an 'Is-Reply' field to the header to indicate if the email contained quoted text. As signatures

are present in a large number of emails and do not carry any special significance from an information standpoint, we also remove user signatures from the dataset using Talon, a popular library with classifiers to identify signature lines [108].

3.1.3 Methodology

Using a custom-built python tool, we analyze the *replies* in all three datasets and compute the amount of information in the *replies* that is already present in one or many past emails. A large amount of repeated content in the *replies* indicates the potential for an effective suggestion mechanism in *reply* construction. We use a custom-built python tool for the analysis of all three datasets. For each email account, the tool calculates the *similarity* between every *reply* and every other email with a timestamp earlier than that of the *reply*. The tool first converts the email text to lower case and removes any punctuation. Then, it deconstructs each email into a vector of words. Stopwords, i.e., words that commonly occur in English but do not have any special meaning like 'a', 'an', 'the', etc. are filtered out from this vector of words. [5] Each word in the vector of words is stemmed to its root. For example: 'presenting' and 'presented are stemmed to 'present'. A concept in English can be expressed using different sentences. For example, 'When is your presentation?', 'When are you presenting?' carry the same meaning, even though the concept is expressed in different words. In this example, after stemming and filtering the stopwords, both the sentences will have the same words. The tool also maintains the number of emails in which a particular stemmed word occurs.

3.1.4 Metrics:

A metric that measures the amount of information in one email (say *em*1) that is repeated in another email (say *em*2) should be - (a) high if a large portion of information in *em*1 is present in *em*2 and vice versa; (b) independent of any other information present in *em*2; (c)

[5] We use the list of popular stopwords in English from Natural Language Tool Kit. [109]

Table 3.3: Example matching email snippets for a user in ENRON dataset

Similarity	Matches
0.9	**reply:** Ken Lay has approved the attached expense report for Rosalee Fleming **match:** I have approved the attached expense report for Rosalee Fleming
0.65	**reply:** As was earlier announced, we will be bringing all Managing Directors together on a quarterly basis. Please note on your calendars the first Monday of every quarter from 8:30 a.m. to 12:00 noon for this purpose. The first meeting will take place on Monday October 1st. If you have any questions, please call Joannie Williamson. **match:** As announced earlier, we will be bringing all Managing Directors together, on a quarterly basis. Please hold open the first Monday of every quarter (from 8:30 a.m. to 12:00 p.m.) for this purpose. However, our first meeting will be on Tuesday, October 2nd.

consider the relative importance of information, i.e. the effect of words that are repeated frequently in several emails should be less than that of special words that occur infrequently. Based on the desired properties stated above, we define the similarity between two emails $em1$ and $em2$ as follows[6]:

$$similarity(em1, em2) = \frac{\sum_{w \in WV_1 \cap WV_2} IDF(w)}{\sum_{w \in WV_1} IDF(w)}; \quad IDF(w) = 1 + log(\frac{N}{C(w)}) \tag{3.1}$$

where WV_1 and WV_2 are lists containing the stemmed words in $em1$ and $em2$, respectively. N is the total number of emails and $C(w)$ is the number of emails containing word w. In other words, the similarity between two emails is defined as the weighted ratio of the number of words common to both the emails to the number of words present in the first email. Each word's weight is a function of the number of emails it occurs in, called the inverse document frequency function IDF. The value of IDF for frequently occurring words is less than that of words that are relatively less common. This metric is also independent of the size of $em2$.

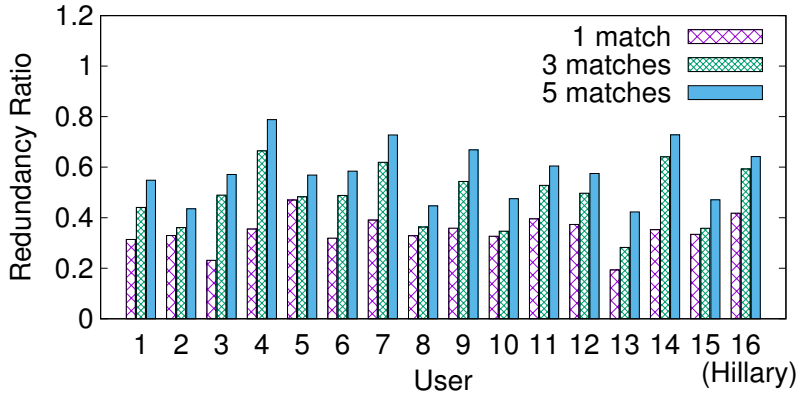


Figure 3.1: Reply redundancy in ENRON dataset and HILLARY dataset

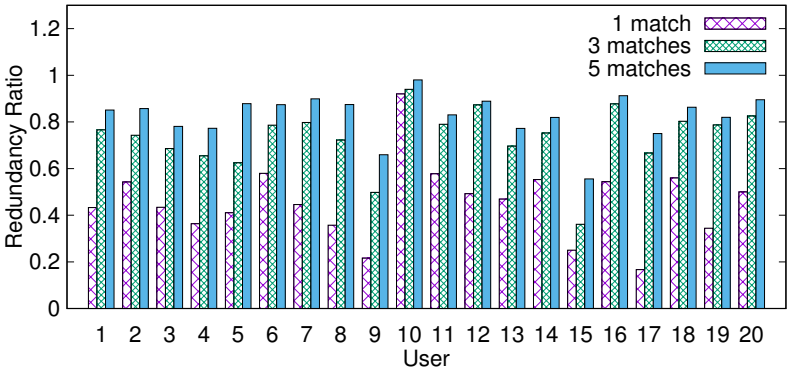
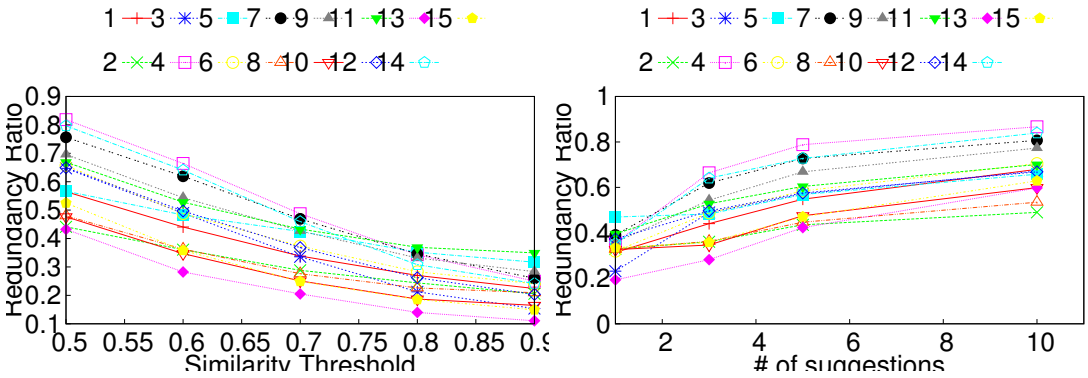


Figure 3.2: Reply redundancy in AVOCADO dataset



(a) τ **(b) Suggestions**

Figure 3.3: Sensitivity analysis for ENRON

3.1.5 Analysis

Using the tool described in the previous section, we first compute the similarities between every *reply* and every other email with a timestamp earlier than that of the *reply* and extract 1/3/5 matching emails having the highest similarity with the *reply*. The amount of *redundancy* in the *reply* is defined as $similarity(em, \sum_{i=1}^{M} m_i)$, where $m_i, \quad \forall i = 1 : M$ are the top M matches for em. We then compute $RedundancyRatio(\rho)$ as the ratio of replies with $redundancy > \alpha$. We choose a threshold (α) of 0.6 as the goal of these experiments is not to find identical matches for the *replies*, but to find emails that match a considerable portion of the *reply*. We evaluate the effect of threshold α later in this section.

Figure 3.1 and Figure 3.2 show the ρ for the ENRON and AVOCADO datasets for 1/3/5 matches. The ρ for HILLARY is included in Figure 3.1 as user 16.[7] For the ENRON dataset, the average percentage of replies with high redundancy (>0.6) was 33.84%, 47.37%, and 57.43%, respectively, for the top one, three and five prior email matches. For the AVOCADO dataset, the average percentage of replies with high redundancy (> 0.6) was 45.80%, 73.46% and 79.87%, respectively, for one, three and five matches. These results indicate that there is a considerable amount of repeated content in all the email accounts.

Table 3.3 shows two examples of *replies* and their corresponding top match from the mailbox of one user in the ENRON dataset. In the first example, the *reply*, and the matched email contained the approval responses to different expense reports for an employee. In the second example, the information that Kenneth Lay is the chairman and CEO of Enron Corp. is present in both the *reply* and the matched email. The *reply* from the second example has a meeting announcement that matched with a similar meeting announcement sent out in the past. These examples illustrate ways in which content is repeated in emails.

[6]Note that the traditional TF-IDF metric does not satisfy (b)

[7]Since the HILLARY dataset is preprocessed without headers or quoted text, there was no way of determining which email was a reply. Here, we computed the redundancy values for all sent emails. The presented results will thus be a lower bound.

Figures 3.2a and 3.2b show the effect of changing the similarity threshold α and number of suggestions, respectively, on the ρ for 15 users in the ENRON dataset. The results for the AVOCADO dataset are similar. As similarity threshold α is increased, the ρ falls for all the users. This is because when α is increased, the threshold at which we decide whether suggestions are useful or not increases. On an average, increasing α from 0.6 to 0.7 decreases the ρ by 15.48%. On the other hand, decreasing α from 0.6 to 0.5 increases the ρ by 18.01%. Also, as expected, as the number of suggestions increases, the ρ increases. Initially ρ increases rapidly, then it saturates. This indicates, the text in the reply is only concentrated in a few emails in the mailbox and is not spread out across a large number of emails. Specifically, as the number of suggestions is increased from 3 to 5, the ρ increases by 31.26% for the ENRON dataset. On the other hand, decreasing the number of suggestions from 3 to 1 decreases the ρ by 26.72%. Increasing the suggestions beyond 10 has little effect on the ρ. From this figure, it can be observed that three would be an ideal number of suggestions as it is around the midpoint of the knee of the curve. We also measured the sensitivity of ρ to the inbox size, sent box size and the number of lines in the reply and found that as the inbox/sent box sizes or the number of lines increases, the ρ increases. The results are omitted in the interest of brevity.

On a Quadcore 3.4GHz Linux computer, finding top 3 matches in a database of approximately 15K emails took 43.7 seconds on average. This indicates that for larger email accounts, finding matches solely on a resource-constrained smartphone might be prohibitive. This motivates an architecture wherein heavy computation related to matching and text processing is done on a cloud and the results pushed back to the email client.

3.1.6 Insights

The analysis has led us to the following key insights - (a) The high degree of redundancies in the replies show that the information in the reply is most likely present in the email account in some form and this can be leveraged to reduce user effort on email and hence

increase productivity; (b) As the number of suggestions increases, the chances of finding the email with similar content increases. If the content of a reply exists in previous emails, it is concentrated in just a few emails. (c) The ideal number of suggestions and the ideal similarity threshold are 0.6 and 3, respectively; (d) Text processing involved with mining the database is computationally intensive and cannot be done only on the mobile device.

It would take a lot of effort for the user to manually search her Inbox and retrieve relevant information, copy it and send a response out. The high degree of redundancies in the replies show that the information in the reply is most likely present in the Inbox in some form and that this can be leveraged to reduce user effort on email and hence increase productivity. For a mobile enterprise user, typing emails on her smartphone is especially cumbersome due to the small real estate on the screen.

3.2 The DejaVu Solution

In this section we present details of *Dejavu*, an automated approach to generation of suggestions to assist in *reply* construction.

3.2.1 Problem Definition and Scope

We define the informational reply suggestion problem as follows - *For an email user, given that a reply to an Inbox email may consist of content that is present in a prior email, can appropriate information be retrieved from earlier emails and provided as suggestions to the user while the reply is being constructed?*[8]. Specifically, the reply suggestion solution should - (1) suggest relevant content and should have high similarity to the intended reply; (2) be presented unobtrusively on a smartphone; (3) should not place a severe burden on the smartphone's constrained resources; (4) be user-friendly and easy to learn; The previous section illustrates that there is considerable amount of redundancy in the content of the sent email.

[8]We do not consider email attachments, something that would be of obvious use to consider. We defer such consideration to future work

Solution Scope

An intuitive way of approaching the problem of generating informational suggestions is to consider the Inbox email as a proxy for the reply, find other emails in the user's mailbox with a large number of words in common as the Inbox email and present them as suggestions. Such a simple solution ($Dejavu_{SIMPLE}$ shown in Figure 3.4) does not consider the fact that emails are unstructured and non-standardized. Several email servers and clients represent the email header and body in their own proprietary formats. Our analysis of the datasets revealed the following issues with email data - (i) Some email clients do not critical fields in email headers like the '$In-Reply-To$'; (ii) Some email clients duplicate the message in different encodings (HTML, Plaintext); (iii) Original text of the email is appended to the reply in non-standard formats; (iv) Users add signatures in different formats. These issues can be solved with the addition of a preprocessing pipeline described in Section 3.2.2. An overview of the simple approach (called $Dejavu_{SIMPLE2}$ with a preprocessing pipeline is shown in Figure 3.5. $Dejavu_{SIMPLE2}$ also suffers from several issues resulting from the usage of natural language in emails: (i) There are noise words in sentences that do not carry any significance (E.g., the, an, at, etc.); (ii) Words of the same semantic meaning are represented in different formats due to inflexion (E.g., meeting and meet); (iii) All words are given equal importance in a sentence. In the rest of this section, we describe *Dejavu*, an automated informational reply suggestion system that overcomes these challenges.

3.2.2 The DejaVu solution

At a high level, *Dejavu* consists of a *Information-Curator* that constructs an *Information Database* with the user's mailbox and indexes it. When the user wants suggestions for constructing a *reply* to an email, the *Information-Curator* extracts context from this email, computes suggestions from the *information database* using the context. When the user wishes to construct a *reply* to an email *em* in her inbox using the *Dejavu* system, the

Considerations:

Should find emails from the mailbox with content closest to the intended reply as suggestions

Design Elements: *Present emails closes to email as suggestions*

- Use the *email* as a proxy for the *reply* to find emails closest to the *reply* within the *Mailbox*
- The number of words in common indicates the 'closeness' between two emails

Pseudocode:

```
buildDatabase (Mailbox):
  FOR i IN Mailbox DO:
    DB.add(id(i), words(body(i))
  RETURN DB
getSuggestions(email, DB):
  FOR entry IN DB DO:
    score = |words(email) ⋂ entry.words|
    scores[id] ← score
  RETURN top(scores)
```

System Considerations:

- At setup time, construct an information *database* using the emails within the *Mailbox* for efficient searching
- Match *email* with every *entry* in the database
- Return the top few *entries* with the highest number of words in common with *email*

Figure 3.4: An overview of *De javu*$_{SIMPLE}$

Considerations:

Should convert unstructured email data into structured database and handle different email formats

Design Elements: *An empirically designed preprocessing pipeline*

- A quotation filter that strips original message content appended to a reply
- Generate missing header fields
- A classifier to remove signature lines appended at the end of an email

Pseudocode:

```
preprocess (email):
  msg ← plaintext(email)
  text, quotes ← quotationFilter(msg)
  body ← removeSignature(text,sender)
  IF quotes:
    parent ← getParent(quotes)
    RETURN body,parent
  ELSE:
    RETURN body,null
```

Systems Considerations:

- Rules for the Quotation filter designed with empirical observations from the AVOCADO and ENRON users
- Missing header value *In-Reply-To* filled by the *ID* of the best match to the stripped quotes in the database

Figure 3.5: An overview of *De javu*$_{SIMPLE2}$

following sequence of events takes place. A user with *Dejavu* client hits reply to an email *em* in her inbox through her email client, the following sequence of operations take place:

- The user selects reply button on the client for the *email*;
- The *Information database* is queried by the *Information-Curator* using the context extracted from *email* and the best matches are retrieved
- The text from these matches is presented to the user at a sentence level granularity as suggestions for the *reply* to *email*.
- When the user selects one or more of the suggested sentences, they are pasted on to the *reply*.
- The user can finish constructing the *reply* by editing the suggested sentences and sends out the *reply*.

In the rest of this section, we describe the key design elements of *Dejavu*.

What is the granularity of suggestions?

Dejavu considers a full email to contain the lowest granularity of stand-alone information, independent of other emails. *Information-Curator* parses emails in their entirety and stores them in the *Information Database*. Therefore, the granularity of suggestions is also full emails. We make this design choice as opposed to other granularities such as sentences because the amount of information (keywords) present in a sentence is low. A sentence is usually not independent but depends on other sentences around it. By considering just a sentence, we may lose out on the surrounding context of that sentence, thus compromising on the appropriateness of the suggestions presented to the user.

What information is stored in the database?

Information-Curator of the *Dejavu* system parses each email from the user's mailbox irrespective of the folder it is in. It separates the email header from the MIME message and

filters out any content that is not plain text, such as attachments, pictures, HTML, etc. Any quoted text (original email attached to a reply) is then removed from the from the email body using a set of rules described in Algorithm 1. Email clients do not follow a fixed standard for attaching quoted text. Therefore, we define these set of rules by extensively observing the format of emails from different clients. The parser for the quoted text looks for specific text patterns indicative of the quoted text and filters out these lines. The resultant email is then scrubbed of any signature lines. Many users set up their email client to attach signatures at the end of every sent email. The format and the content of these signatures vary among different users and sometimes, for the same user. We, therefore, use a signature extraction tool called Talon [108], which determines whether a line is a signature line or not by extracting a set of features from that line (for example, the presence of words such as 'best', 'thanks', 'regards', etc.) and passing it through a Support Vector Machine (SVM). This SVM is pre-trained on a large annotated email corpus.

Apart from the email body, the *date* the email was sent/ received, the *ID* and the *subject* are also extracted from the email. Modern email headers have an 'In-Reply-To' field for *replies* that contains the *ID* of the (parent) email to which the current email is the *reply*. *Information-Curator* collects this parent email message *ID* from the header. Any remaining lines that do not have any quoted text or signatures are added to the *Information Database* along with the *ID*, the parent email *ID* (if any), the *date* and the *subject*.

How is the content indexed?

Each entry in the *Information Database i* is indexed by a set of keywords extracted from it. The text of an entry in the *Information Database* is initially converted to lowercase and then split into constituent words $W(i)$. Any punctuations are removed from these words. The most common words in English, also called 'stopwords' are then filtered and removed from $W(i)$, as the presence or the absence of stopwords does not carry any lexical significance when it comes to the extracting the core context of an email. Table 3.4 shows a list of the

Table 3.4: List of stopwords filtered by *Dejavu*

i, me, my, myself, we, our, ours, ourselves, you, your, yours, yourself, yourselves, he, him, his, himself, she, her, hers, herself, it, its, itself, they, them, their, theirs, themselves, what, which, who, whom, this, these, those, am, is, are, was, were, be, been, being, have, has, had, having, do, does, did, doing, a, an, the, and, but, if, or, because, as, until, while, of, at, by, for, with, about, against, between, into, through, during, before, after, above, below, to, from, up, down, in, out, on, off, over, under, again, further, then, once, here, where, why, how, all, any, both, each, few, more, most, other, some, such, no, nor, not, only, own, same, so, than, too, very, s, t, can, will, just, there, when, that, don, should, now

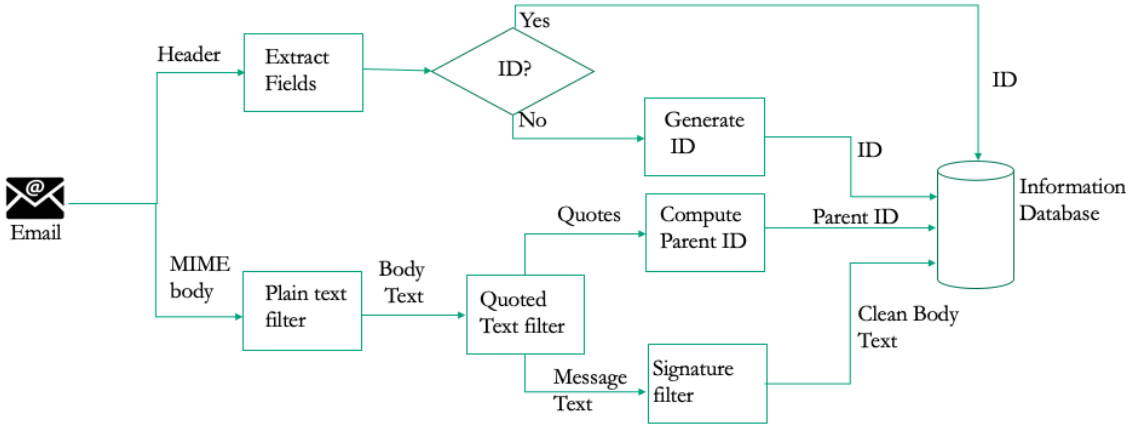


Figure 3.6: Pre-processing pipeline in *Dejavu*

stopwords that are filtered by the *Information-Curator*.

The remaining words in $W(i)$ are then stemmed to their roots. Several words in English are derived from the same root by the annexation of suffixes and prefixes. For example, the words 'addition', 'additive', 'adding' are derived from the root 'add' through suffixes '-ition', '-itive', '-ing', respectively. All of these words have a meaning close to that of 'add'. To capture the core context of the text in the *index* and to avoid duplicates of words that are close in meaning to each other, we trim every word in $W(i)$ to its root. Each entry in the *Information Database* is then indexed on the set of roots of words in $W(i)$. Figure 3.6 summarizes the preprocessing pipeline of *Dejavu*.

How are suggestions extracted?

In *Dejavu*, the suggestions are extracted using a keyword matching algorithm. The email whose suggestions are to be extracted (*em*) is parsed, and the core context in the form of a list of keywords $KW(em)$ is extracted from it. The *Information-Curator* then matches $KW(em)$ with the *index* of entries in the *Information Database*. An obvious solution for finding suggestions would be to match $KW(em)$ with just the keywords in the *index* of an entry in the *Information Database*. However, this simplistic solution will most likely not work well in the context of emails. This is because email, apart from being a method for information sharing, is primarily an asynchronous medium of communication between two parties. This is because email is primarily designed as a medium of asynchronous communication between two parties. Most of the emails are conversations between individuals and the context of one conversation might not be contained entirely within one email and can span multiple emails. Consider the following example of an email - *Where do you live?* and the corresponding *reply* - *On Mars*. For this example, the information contained in the *reply* does not hold much significance on its own. The email provides context for the *reply*. Therefore, an email and the *reply* when considered together carry significance, and not separately. Also, given the rising trends in the usage of email on mobile devices, users often resort to shorter replies and informal sentence construction (like the *reply* in the example above). In this case, without the parent email's *index*, it would be hard to retrieve any information relating to the conversation, just from the child email's context. Therefore, *Dejavu* combines the keywords in the indices of an entry and its parent (if any) to find matches i.e. $KW(em)$ is matched with $index(i) \cup index(parent(i))$.

The degree of match (*similarity*) between a set of keywords KW and the combined index $cindex(i) = index(i) \cup index(parent(i))$ is computed as $\frac{\sum_{w \in KW \cap cindex(i)} IDF(w)}{\sum_{v \in KW} IDF(v)}$, where IDF is the inverse document frequency function defined in Equation 3.1[9]. In other words, *similarity* is the ratio of the sum of IDF for words that are present in both the set of key-

[9]This is the same as the right hand side of Equation 3.1

words KW and the combined index $cindex(i)$ to the sum of IDF for all the keywords in KW. Using IDF as weights in the ratio for keyword matching ensures that the presence/absence of keywords that occur less frequently in the user's mailbox carries a higher weight in computing the *similarity*, compared to keywords that are relatively more common. This is based on the intuition that keywords that occur with less frequency carry more importance. The weight is a function of the inverse of the frequency at which the keyword occurs in the database. We choose a bag-of-words similarity metric over other neural metrics like word mover's distance, paragraph2vec similarity to remain immune to lower volume of training data and the possibility of out of vocabulary words.

After computing the *similarity* between KW and every other entry *i* in the *Information Database* , *Information-Curator* then returns a set of information entries with the highest *similarities* to *em* as suggestions to the email [10].

When are the suggestions retrieved?

The *Dejavu* client uses a hybrid push/pull model for retrieving suggestions. Upon receiving a new email *em* in any folder of the mailbox, the *Information-Curator* computes suggestions $S(em)$ and stores them in a *Suggestion Database*. $S(em)$ is pushed to the *Dejavu* client on the smartphone, who stores it in a local database. This database on the smartphone only stores the suggestions for a small fixed number of recent emails (say 100). When the smartphone user hits 'reply' to an email, the suggestions are retrieved from the local database by the *Dejavu* client and presented to the user. If the suggestions for the email are not already present on the local database, the *Dejavu* client pulls them from the *Information-Curator*. Storing a copy of suggestions on the *Dejavu* client enables the smartphone user to retrieve suggestions even when she is offline and not connected to the *Information-Curator*.

[10]Note that while the combined index of an entry and its parent is used in matching, only the text of the entry is included in the suggestions.

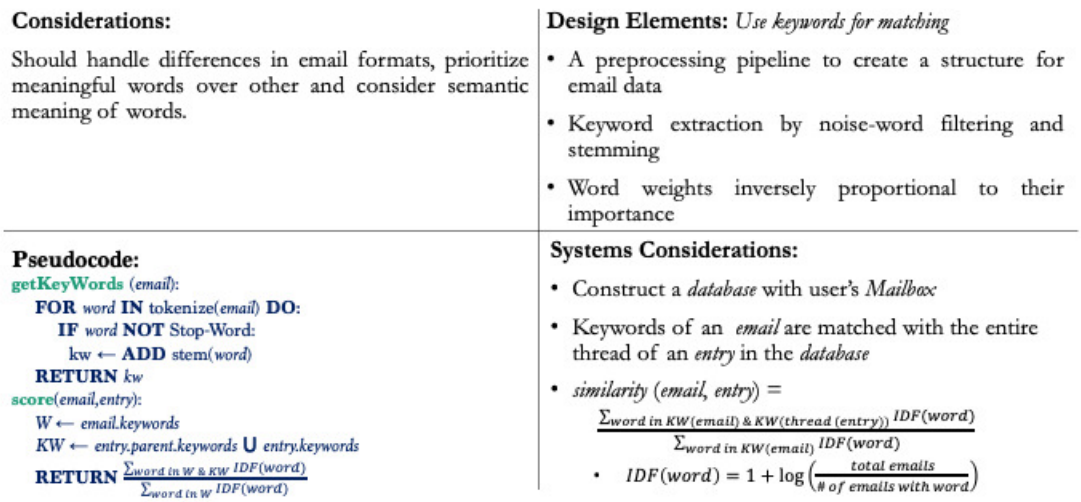


Figure 3.7: An overview of *Dejavu*

How are the suggestions presented?

When the user selects 'reply', a list of constituent sentences in a suggestion grouped by their *subject* lines are shown to the user. The user can select any number of these sentences. Upon selection, these sentences are automatically copied onto the clipboard and pasted during reply construction.

An overview of *Dejavu*'s keyword matching algorithm is shown in Figure 3.7.

3.3 *Dejavu++*: Optimizations to *Dejavu*

The baseline *Dejavu* algorithm uses keyword matching with Inverse Document Frequency (IDF) weights to find suggestions for an inbox email. We later show in Section 3.4 that the baseline algorithm succeeds in presenting useful suggestions to an email 27% of the time on average. Recall that *replies* in a user's mailbox have a significant redundancy ratio - 60.4%. Baseline *Dejavu* is only able to leverage 45% of this redundancy through suggestions. In this section, we identify the following issues with *Dejavu* that prevent it from leveraging the considerable redundancy present in a user's *replies*. We later propose *Dejavu++*, a suite of optimizations to the baseline *Dejavu* to overcome these issues.

- For every Inbox email, the baseline algorithm searches the entire database for possible suggestions. A degree of match (similarity) is computed for each entry in the Information Database having at least one keyword in common with the Inbox email. For heavy email users with large mailboxes, this simple search results in a heavy computational burden. For these users, it is possible that the suggestions for an email may not be presented at the smartphone in time for the user to include them in her *replies*.

- *Dejavu*'s baseline algorithm does not leverage any other information within the user's past email history beyond email body keyword matching. The algorithm misses crucial information such as past matching performance, the sender/receiver characteristics, the timing characteristics, etc.

- For every email, baseline *Dejavu* computes the suggestions only once when it arrives at the user's Inbox. *Dejavu* does not refresh these suggestions when new information becomes available as the user types the *reply*.

- Baseline *Dejavu* relies on user's own mailbox for computing suggestions. As employees communicate with other employees within the enterprise, crucial information relating to their job functions may be located with other employees' mailboxes within the enterprise. Baseline *Dejavu* misses out on the information outside the user's mailbox.

In the rest of this section, we present *Dejavu++*, an optimized version of *Dejavu*'s automated response suggestion system. *Dejavu++* retains the core of the matching algorithm of *Dejavu* but includes several enhancements to address the issues outlined earlier.

3.3.1 Reduction of Computation Complexity With Topic Filters

Most enterprise employees responsibilities involve several different functional roles at the same time. For these employees, the content of email correspondence related to one job

function will be different from the content related to other job functions. It is unlikely that the *replies* corresponding to one job function have any content in common with *replies* corresponding to other job functions. For example, consider the case of an account manager at a mid-size retail company whose responsibilities include three different business functions - handling stock purchases from vendors, running employee payroll and approving travel expenditures. Her emails to clients relating to stock purchases would be different nature compared to emails to her colleagues regarding payroll inquiries. *Dejavu++* uses this insight to reduce the complexity of retrieving suggestions. *Dejavu++* groups the emails present in a user's mailbox into several categories and computes suggestions for an Inbox email by *matching the keywords of the email only to entries from the information database grouped within the same categories as the email*. By reducing the set of possible candidates for suggestions, *Dejavu++* improves the speed at which suggestions are computed.

We now present the details of category-wise filtering of the information database as a series of challenges and design choices:

How are the categories defined for a user?

Dejavu++ automatically obtains the hidden categories within a user's mailbox through *topic modeling*. Topic modeling refers to a class of supervised natural language processing techniques to discover hidden topics within a large corpus of unstructured data [110–112]. These techniques observe word co-occurrences within documents of the corpus and group words into topics. Given a new document, the words of this document indicate the topics present in that document. For example, given a corpus of documents containing only academic papers on mobile computing and news articles on politics, topic modeling of this corpus would infer that words such as 'abstract' or 'wireless networks' belong to a topic (say $topic_1$) that is different than words such as 'Senate', or 'President' (say $topic_2$). Given a new news article on the Senate elections, these techniques can infer that the article belongs to $topic_2$.

Specifically, *Dejavu++* uses Latent Dirichlet Allocation (LDA) [110] topic models and Gibbs sampling to automatically extract the latent set of topics (or categories) from a user's mailbox. Keeping in line with the document generation process of LDA, *Dejavu++* assumes that each entry in the information database, i, is generated with the following story:

- Each entry, i, contains a probabilistic combination of K topics. The value of K is empirically fixed for the user.

- Each one of the K topics is a probability distribution ϕ_k over the set of words in the vocabulary V. ϕ_k for each topic k is a vector of $|V|$ probabilities that sum to 1. The probability distribution ϕ_k has a Dirichlet prior λ. The probabilities for words more likely to be associated with the topic k are higher in ϕ_k than others.

$$\phi_k \sim Dirichlet(\lambda)$$

- Each entry, i, is a distribution θ_i over the K topics. θ_i is a vector of K probabilities that sum to 1. The probability distribution θ_i has a Dirichlet prior α. In θ_i, the probabilities for topics associated with the entry i are higher than other topics.

$$\theta_i \sim Dirichlet(\alpha)$$

- If the entry i has n_i words, the topic assignment $z_{n,i}$, for the word $w_{n,i}$, is obtained from θ_i.

$$z_{n,i} \sim Multinomial(\theta_i)$$

- Finally, the n^{th} word in the entry i is selected from the vocabulary according to the probability distribution $\phi_{z_{n,i}}$.

By assuming this generative story and observing the result of the generative process, i.e., words w_n within each entry i, *Dejavu++* infers the topic-word probability distribution

ϕ_k and the document-topic probability distribution θ_i using techniques like Gibbs sampling [113]. This estimation occurs periodically once every few days at the *Information-Curator*.

How to categorize the Information Database?

Dejavu++ trains on a user's information database to obtain the topic-word probability distribution ϕ_k for all of the K topics and the document-topic probability distribution θ_i for all of the entries i. Within the document-topic distribution of an entry i, the probabilities corresponding to topics most likely to be present in the entry i are higher than other topics. *Dejavu++* sorts the distribution θ_i in decreasing order of probabilities and obtains L topics, T_i, having the highest probabilities.

$$T_i = k|\theta_i^{(k)} \geq X_{-L}, \qquad X_{-L} = max(t|\#\{k \in K|\theta_i^{(k)} \geq t\} = L)$$

Dejavu++ assumes the entry i contains these T_i categories and labels it with the topics T_i. For easy retrieval using the topic labels, *Dejavu++* also creates an index for the database on these labels.

How to use the categories to compute suggestions efficiently?

To compute suggestions for an email *em*, *Dejavu++* analyzes the words in the email *em* and estimates the posterior document-topic probability distribution θ_{em} for *em*. From this distribution, *Dejavu++* obtains the top L topics (with highest probabilities), T_{em}. *Dejavu++* matches the keywords present in the email, $KW(em)$, with the combined index of each entry, $cindex(i)$, in the information database, labeled with at least one of the topics present in the set T_{em}. The combined index *cindex* is defined in Section 3.2.2. The top few entries with the highest degree of match are presented as suggestions to the user.

By matching the keywords of an email with only those entries having the same latent topics as the email, *Dejavu++* reduces the complexity of retrieving suggestions. An

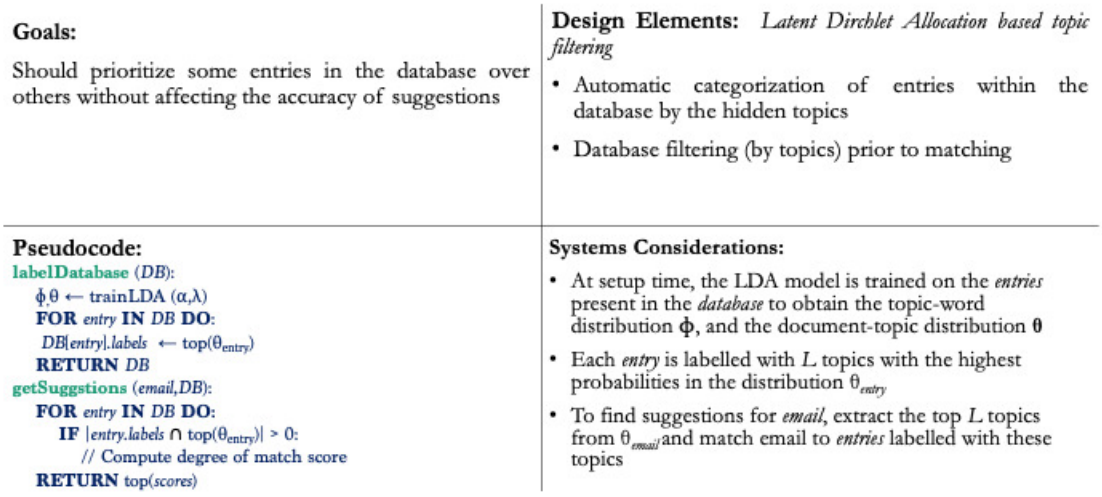


Figure 3.8: An overview of topic filtering in *Dejavu++*

overview of *Dejavu* with Topic filtering is shown in Figure 3.8.

Improving the accuracy of suggestions with scoping

The baseline *Dejavu* algorithm matches the keywords of an Inbox email *em* with the indices of every candidate entry (with at least one matching keyword) within the information database to retrieve suggestions. During matching, each entry in the database is equally likely to be a relevant suggestion. However, user's email habits are subject to certain patterns which the naive matching algorithm doesn't take into account for finding suggestions.

Consider the case of an enterprise user Alice, who is a field sales representative at a large global 500 company. Alice's mailbox contains email correspondence relating to sales with her team, monthly budget reports to her manager, travel arrangements to client locations with the travel team, miscellaneous other work-related and personal topics. Alice's responses will have different characteristics depending on the query within the email, the person she is responding to, the time of the day, the subject matter of the conversation, etc. For example, Alice's responses will usually be longer if it is a monthly budget report to be sent to her manager, than a response to the travel team approving her expenses for the recent client visit. She may type longer responses at her desk during early hours of

a day than when she is at a client location mid-day. She may respond faster to a critical work-related query from a client than a personal query. In addition to patterns within Alice's response construction, there will also be patterns in the redundancy (as explored in Section 3.1) within these responses. For example, A query from Alice's colleague asking for the monthly budget report will be redundant with the response containing the budget report sent out her manager earlier. Responses to queries from her manager on the status of a sales contract might have content already present within the email correspondence with the client of the sales contract. Responses to work-related queries might have significant redundancies with email correspondence during the day (as opposed to the night). *Dejavu*'s matching algorithm doesn't take into account the email behavior of Alice when computing suggestions.

Dejavu++ utilizes these patterns to retrieve more relevant suggestions for an Inbox email *em*. *Dejavu++* assumes that the features of an email *em* are related to certain features of it's response r_{em}. While the relationships between these features can exist across multiple dimensions, we limit our focus to three major features of a response - (i) The length of the response ($L_{r_{em}}$), (ii) The time to respond since the email *em* was received ($T_{r_{em}}$), and (iii) the senders/recipients of emails with significant redundancy to the response ($E_{r_{em}}$). To retrieve suggestions for an Inbox email *em*, *Dejavu +* estimates three features $L^p_{r_{em}}$, $T^p_{r_{em}}$ and $E^p_{r_{em}}$ using *em*. Specifically, to estimate the length and the time to respond, *Dejavu++* uses *em*, r_{em} pairs within the user's mailbox to train a regression model with input features of the input email *em* such as the length L_{em}, the time of day of arrival T^d_{em}, the day of week of arrival T^w_{em}, the senders/recipients E_{em}, the subject keywords KW^S_{em} and the keywords within the email body KW_{em} to output a target $L_{r_{em}}$ and $T_{r_{em}}$. To estimate the sender/recipient set, *Dejavu++* uses the same set of input features to train a classifier to output '1' if a particular email address is present in $E_{r_{em}}$ and 0 otherwise.

It matches the keywords of *em* with those entries in the Information Database within a margin of error of the predicted features. If ε_L, ε_T and ε_E the margins for the length of

response and time to respond, *Dejavu* + matches keywords of *em* - $KW(em)$ with all entries em' in the database whole length $L_{em'}$, timestamp $T_{em'}$ and sender/receiver set $E_{em'}$ satisfies the following conditions - (i) $L^p_{r_{em}} - \varepsilon_L < L_{em'} < L^p_{r_{em}} + \varepsilon_L$; (ii) $T_{em} + T^p_{r_{em}} - \varepsilon_T < T_{em'} < T_{em} + T^p_{r_{em}} + \varepsilon_T$; and (iii) $|E^p_{r_{em}} \cap E_{em'}| = \varepsilon_E$

3.3.2 Improving the relevancy of suggestions with user feedback

Baseline *Dejavu* computes suggestions for an Inbox email only once when it is received in the user's mailbox, using only the keywords present in the email. While this strategy can indeed reduce the burden of typing replies if the user uses the suggestions, it fails to consider any potential user feedback that becomes available if the user does not incorporate the suggestions when constructing a *reply*. User feedback can either be (i) explicit when the user performs an action to inform the system that the suggested *replies* are not useful; or (ii) implicit, when the user pulls up suggestions but does not incorporate them in the *reply*. *Dejavu++* improves upon *Dejavuby provisioning for both implicit and explicit user feedback to recompute suggestions when the initial set of suggestions are deemed irrelevant by the user.*

We now present the details of how *Dejavu++* leverages the implicit and explicit user feedback to enhance the relevancy of suggestions.

When are the suggestions recomputed?

Dejavu++ updates the initially computed suggestions when it receives new information on the usefulness of the presented suggestions through feedback from the user. *Dejavu++* infers that the presented suggestions were not relevant if the user looks at the suggestions (by clicking on the suggestions option from the email menu) and proceeds to type the *reply* without incorporating the suggestions. In addition to monitoring user's behavior for implicit feedback, *Dejavu++* also allows the user to explicitly mark the given suggestions as irrelevant and ask for new suggestions with a refresh button. *Dejavu++* recom-

putes the suggestions for an email under the following circumstances: (i) the user clicks the refresh button on the suggestion panel; (ii) the user has types a portion of the reply without using suggestions even after looking at the suggestions panel , and (iii) the user types a portion of the reply without looking at the suggestions panel. Under the circumstances as mentioned above, the *Dejavu++* email client on the user's smartphone triggers the *Information-Curator* to refresh the suggestions.

What feedback is collected from the user?

To compute more relevant suggestions (than the ones initially presented), the *Information-Curator* needs to use additional information beyond just the keywords of the email. Note that for usability reasons, the user cannot be prompted to provide any further input to aid the recomputing process. This information has to be inferred from the natural sequence of actions taken to respond to the email. Given that the presented suggestions $S_{old}(em)$ are already passively marked as irrelevant by the user under certain circumstances, *Dejavu++* leverages them recompute new suggestions. In the case when the user starts to type the response with or without looking at the presented suggestions, the content of the partially typed *reply* itself provides useful clues to recompute the suggestions. Therefore, *Dejavu++* email client collects the partially typed reply $r^{(p)}$ in addition to the old irrelevant suggestions $S(em)$ and sends them to the *Information-Curator* to recompute suggestions.

How to utilize feedback to improve the relevancy of suggestions?

Upon receiving a trigger to recompute suggestions for an email *em*, *Information-Curator* first identifies the circumstance under which the suggestions are to be recomputed. When the user either explicitly taps the refresh button or starts typing a *reply* from scratch after looking at the initial set of suggestions, *Dejavu++* concludes that the suggestions were not useful. These entries comprising the suggestions $S_{old}(em)$ are marked as irrelevant in the information database to exclude them from being considered as potential suggestion candi-

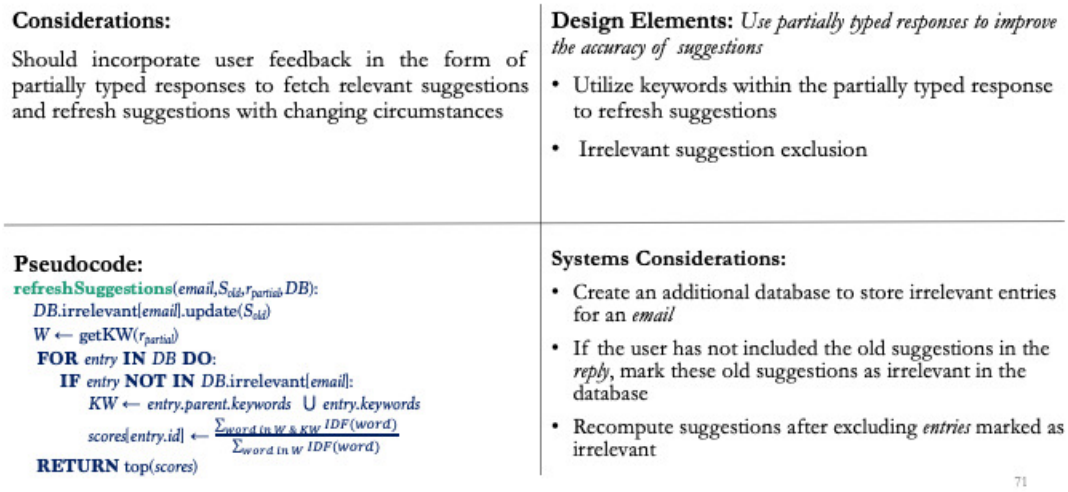


Figure 3.9: An overview of partial reply matching in *Dejavu++*

dates during any future suggestion refreshes. The relevancy of an entry i in the information database is indicated by a temporary bit flag $irel_i$ set to 1 $\forall i in S_{old}(em)$. If a partially typed reply is received from the email client, the *Information-Curator* extracts the keywords from it $KW(r^(p))$, and matches these keywords to the combined index of every entry i in the information database with the flag $irel_i$ set to 1. In the case where the user has not typed a reply yet, the keywords of the Inbox email em are instead matched to the entries in the database with an unset relevancy flag. Similar to *Dejavu*, the top few entries with the highest degree of match score with the keywords (of the partial reply or the email) are presented to the user as suggestions. An overview of *Dejavu++*'s partial reply matching is shown in Figure 3.9.

3.3.3 Expanding the sources of suggestions to the global network of mailboxes

Email is the most popular medium of asynchronous communication within enterprises. The combined set of mailboxes of all the users in the enterprise contains a large amount of information on various business functions within that enterprise. Consider the case of four enterprise employees Alice, Bob, Christy, and Derek. Assume that Christy has compiled a quarterly sales report and sent it over email to Derek (em^c). Let's say Alice sends an email

to Bob asking for the same quarterly sales performance numbers (em^a). Even though the response for em^a is present in within the enterprise network of mailboxes (in em^c), baseline *Dejavu* will not be able to present it as a suggested $reply$, as it only searches for suggestions within a user's own mailbox.

To leverage the vast amount of information present across the enterprise, *Dejavu++* includes the entire enterprise's network of mailboxes in the search for relevant suggestions. When a user U receives an inbox email em, *Dejavu++* initially matches the keywords of em with the combined indices of entries within the user's own Information Database DB_U. If the magnitudes of degree of match scores between the top few database entries with the highest similarity to $KW(em)$ are less than a threshold τ_{nw}, *Dejavu++* proceeds to match $KW(em)$ with entries from other user's information databases $DB(U')\forall U' \in enterprise$. To respect user privacy, users are given an option to exclude their mailbox from the search for suggestions for other users. They can also selectively configure the visibility of their mailboxes individually for users in the enterprise. *Dejavu++* restricts the search to only those mailboxes that are pre-configured to be visible by the user U during suggestion retrieval.

3.3.4 Architecture

Figure 3.10 shows the system architecture of *Dejavu*. There are two main components to *Dejavu*: The *Information-Curator* and the *Dejavu* client.

Information-Curator*:*

The *Information-Curator* of a *Dejavu* system is responsible for maintaining all the relevant databases and for retrieving suggestions from them. It resides in an elastic cloud such as the Amazon EC2 [114] and accepts connections from a *Dejavu* client. It consists of five modules:

- **IMAP Polling:** This module is responsible for frequently polling the user's mail server and downloading any new email content using the IMAP protocol. It acts like

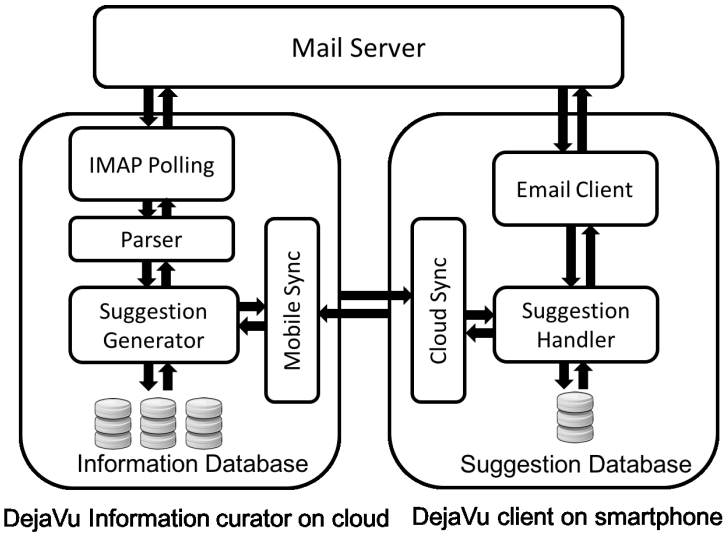


Figure 3.10: System architecture of *Dejavu*

an email client for the user's mail server. The new emails are passed to the Parser module for further processing.

- **Parser:** This module receives emails from the *IMAP Polling* module, parses them to extract relevant information and adds them the database. As discussed earlier, for each email, the body content is filtered to remove quoted text and signature lines. Other information such as *date*, *ID*, parent's *ID* (if any) are also extracted. The parser then computes the *index* for this email and sends the *text*, *date*, parent's *ID*, the *index* and the folder to the *Suggestion Generator*.

- **Suggestion Generator:** If the email does not belong to any 'Sent' folder of the user, the *suggestion generator* retrieves suggestions for this email. The information extracted from an email by the parser along with the suggestions (if any) is added to the database. If the email has suggestions, these suggestions are also sent to the Mobile sync module. If the *Dejavu* client requests suggestions, the *Suggestion Generator* is retrieves them and sends them back.

- **Mobile sync:** This module is responsible for handling communications to and from the *Dejavu* client. It forwards any new suggestions generated by the *Suggestion*

Generator to the *Dejavu* client and handles any requests for suggestions from the *Dejavu* client.

- **Information Database:** This is the core module of *Information-Curator* that stores all the information content of the mailbox. It consists of three databases: Information store, suggestion database, and a word frequency database.

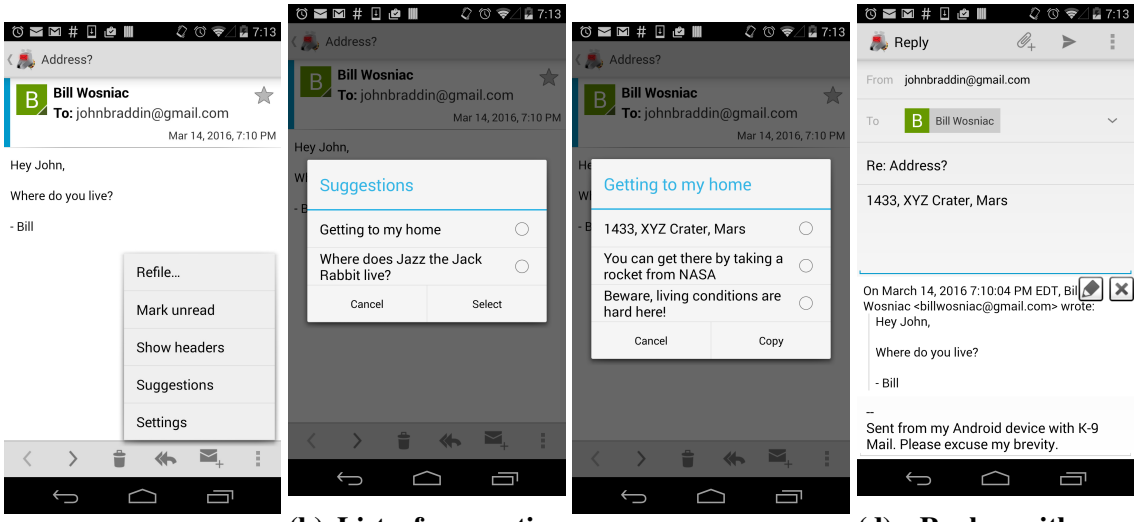
Dejavu *Client:*

The *Dejavu* client is located on the user's smartphone and interacts with the *Information-Curator* on the cloud. The *Dejavu* client consists of four modules:

- **Cloud Sync:** This module receives suggestions from the *Information-Curator* and forwards suggestion requests to the *Information-Curator*.
- **Suggestion Database:** This database stores the suggestions for a fixed number of latest emails from the user's mailbox.
- **Suggestion Handler:** This module is responsible for retrieving the suggestions for an email that the *Email client* requests from the suggestion database. If a suggestion is not present, it is requested from the *Information-Curator* through the *Cloud Sync* module. Any suggestions pushed to the *Dejavu* client from the *Information-Curator* are added to the *Suggestion Database*.
- **Email Client:** This is the component that a user directly interacts with. It has all the functionalities of a typical email client in addition to the suggestion presentation feature. When the user wants to reply to an email, the email client requests the *Suggestion Handler* for any suggestions to that email and presents them.

3.3.5 Prototype

We developed a prototype for the *Dejavu* client on Android OS and *Information-Curator* on a Linux machine (in Python). We modified the source code of K-9 mail client [115],



(a) Suggestions option (b) List of suggestion emails (c) List of snippets (d) Reply with selected text

Figure 3.11: Prototype screenshots

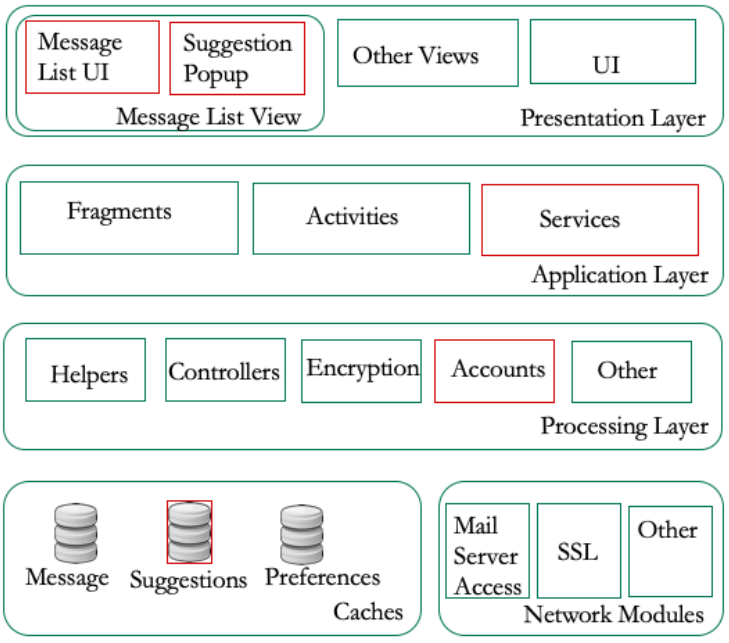


Figure 3.12: Integration with K-9 email client

a popular open source email client application for Android to act as a *Dejavu* client. K-9 mail is a full-fledged email client with functionalities such as search, IMAP push, folder sync, filing, signatures, etc. and supports email access through IMAP, POP3 and Exchange servers. Figure 3.12 shows the modules added or modified within K-9's architecture. The modifications made to the application have a minimal footprint: less than 200 lines of code.

We add a 'Suggestions' options to the UI of an email (see figure 3.11a. When the user selects this option, a list of suggestions for that email, retrieved with options to select them (see figure 3.11c). from a database in the external storage directory, are displayed (figure 3.11b. The subject line of the suggestion is displayed on the list. When the user selects one of these suggestions, another dialog box with a list of constituent sentences is displayed on the screen (see figure 3.11c) with options to select any number of these sentences. When the user hits 'copy', a *reply* is constructed with the selected sentences. The user can edit the *reply* before sending it out (see figure 3.11d).

3.4 Evaluation

3.4.1 Methodology

We evaluated *Dejavu* on 15 users from ENRON dataset and 20 users from the AVOCADO dataset (Section 3.1.1). For each user, the emails in the dataset are processed and sorted in the increasing order of their timestamp.

For the AVOCADO dataset, 75% of the emails from the sorted list of emails are used to populate the *Information Database* in *Information-Curator*. All other emails are then accessed in order. If the email is not a *reply* (no quoted text has been encountered during initial parsing), it is added to the *Information Database*. If the email has a *reply*, the matching algorithm is used to retrieve suggestions. These suggestions are stored in the suggestions database, and the email is added to the *Information Database*. The *reply* text for the emails for this dataset was extracted by looking up *reply_id* in the database. The similarity (equation 3.1) between the *reply* text, and the union of suggestions is calculated.

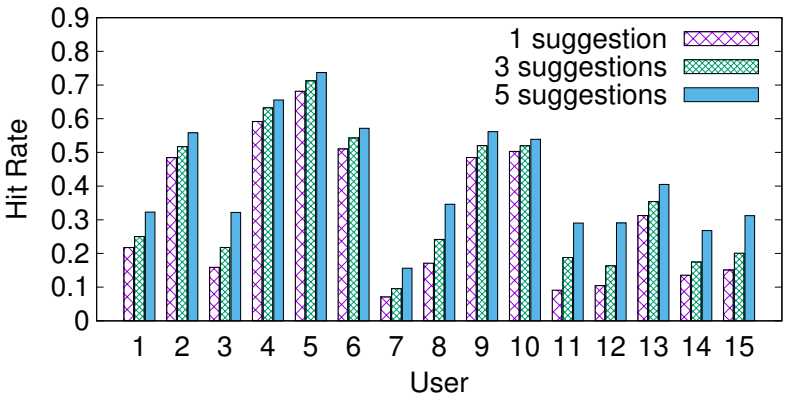


Figure 3.13: Hit rates for *Dejavu* on the ENRON dataset

We process the ENRON dataset differently from the AVOCADO dataset due to the absence of 'In-Reply-To' field in the header. In this case, the *reply* to a specific email cannot be identified by simply parsing email header. To overcome this problem, we keep track of which emails have quoted text. The presence of quoted text implies that these emails are *replies* to some other email. We further process the quoted text for these emails using the same rules as in Section 3.2 to obtain the text of the parent email. The emails are then accessed in increasing order of their timestamp. If the email is not a *reply*, it is added to the database. If the email is a *reply*, the text of the parent email extracted from the quotes is used to lookup the database for suggestions. The similarity between the email text and the union of suggestions (retrieved from the quoted text) is then computed.

To evaluate the suggestion retrieval algorithm, we define a metric $HitRate(\tau)$ for a similarity threshold τ to be the ratio of the number of emails whose *reply* has a *similarity* greater than τ with the suggestions (*n_high*) to the total number of emails with replies. A high value of $HitRate$ indicates that the suggestions were useful in writing replies.

3.4.2 Macroscopic Results

We evaluated $HitRate$ for at threshold $\tau = 0.6$ for the 20 users in AVOCADO dataset (shown in figure 3.14) and 15 users in the ENRON dataset (shown in figure 3.13), for 1, 3 and 5 suggestions. For the AVOCADO dataset, the average $HitRate$ for 1, 3 and 5

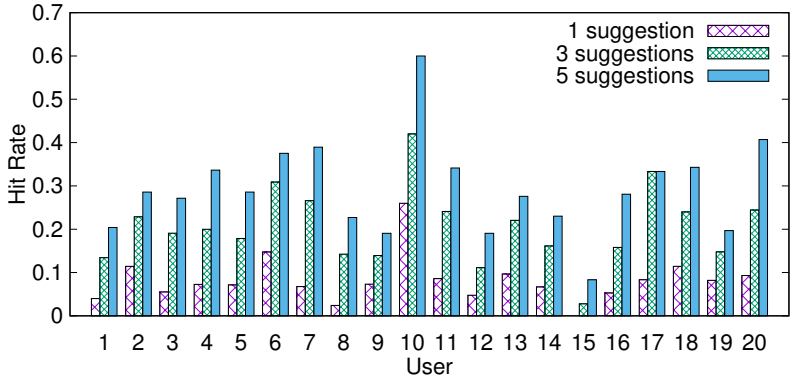
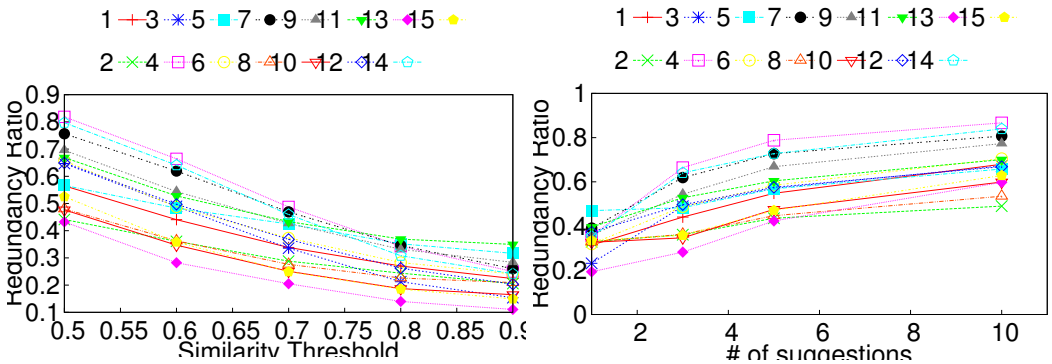


Figure 3.14: Hit rates for *Dejavu* on the AVOCADO dataset

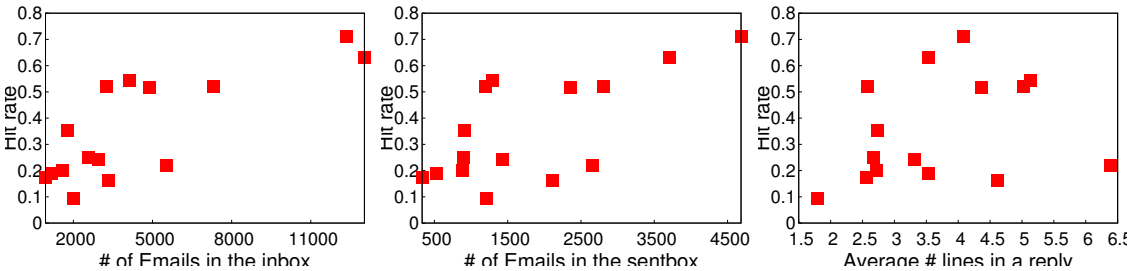


(a) Similarity Threshold **(b) Number of Suggestions**

Figure 3.15: Sensitivity analysis for *Dejavu* on the ENRON dataset

suggestions was 0.08, 0.20 and 0.29 respectively. For the ENRON dataset, the average *HitRates* for 1, 3 and 5 suggestions were 0.31, 0.42 and 0.51, respectively. In other words for the case of 3 suggestions, on an average *Dejavu* was able to retrieve useful suggestions for one in 5 replies for the AVOCADO dataset and 1 in 3 emails in the ENRON dataset.

We also compute the *HitRate* for a fictional suggestion that contains all the keywords seen in the past for that user, and found that it is greater than 0.9 for all the users in both the datasets. This indicates that the replies rarely contain new keywords. These numbers indicate the efficiency of the *Dejavu*'s suggestion algorithm in retrieving useful suggestions for a user.



(a) Inbox size **(b) Sentbox size** **(c) Reply size**

Figure 3.16: Sensitivity to various parameters for *Dejavu* on the ENRON dataset

3.4.3 Microscopic Results

In this section, we evaluate the sensitivity of *HitRate* to various parameters. Figure 3.15a shows the effect of changing the similarity threshold τ on the *HitRate* for 15 users in ENRON. As the similarity threshold, τ is increased, the *HitRate* falls for all users. This is because as τ increases, the threshold at which we decide whether suggestions are useful or not increases. On average, increasing the similarity threshold from $\tau = 0.6$ to $\tau = 0.7$ decreases the *HitRate* by 15.48% for users in ENRON dataset. On the other hand, decreasing the similarity threshold from $\tau = 0.6$ to $\tau = 0.5$ increases the *HitRate* by 18.01% for ENRON users.

Figure 3.15b shows the variation of *HitRate* to changes in the number of suggestions for 15 users in ENRON. In general, as the number of suggestions is increased, the *HitRate* increases. Initially the *HitRate* increases rapidly and then it saturates. This indicates that the text in the reply is only concentrated in a few emails in the mailbox and is not spread out across a large number of emails. Specifically, as the number of suggestions is increased from 3 to 5, the *HitRate* increases by 31.26% for ENRON users. On the other hand, as the number of suggestions is decreased from 3 to 1, the *HitRate* decreases by 26.72% for ENRON. Increasing the number of suggestions beyond 10 has little effect on the *HitRate*. From these figures, it can be observed that three would be an ideal number of suggestions as it is around the midpoint of the knee of the curve.

We also evaluate the sensitivity of *HitRate* to the size of the Inbox, size of the sentbox

and the average number of lines in the reply for the ENRON (shown in Figure 3.16) dataset. However, we only present the results for the ENRON datasets in this chapter. As the inbox size increases, the *HitRate* increases for the ENRON dataset. With larger inbox sizes, there is more information available in the database for lookup, and hence a higher chance of finding the right suggestions for the replies. Also, as the number of emails in the sentbox increases, the *HitRate* increases.

As the replies become longer (number of sentences in the replies increases), *HitRate* generally increases (if a few outlier points are ignored). This is probably because for a more extensive reply there is more scope for a suggestion to be useful. To conclude, in general, larger inbox size, larger sent box size, and larger reply size tends to correlate with a larger *HitRate*. As more and more content is encountered in the mailboxes, the *HitRate* is expected to improve for any user. Table 3.5 shows two examples *Dejavu*'s suggestions. For both these cases, the content of suggestion is very close to the *reply* The first example is a meeting scheduling email sent to an executive, to which the reply is a confirmation email. One of the suggestion snippets for this was a sentence from a confirmation email from another meeting scheduling email from the past, wherein the same executive asks the meeting be put on the calendar. The second example is an email requesting information on the time of a conference call. In this case *Dejavu* was able to pull up an email in the past which contained information on the time of this call successfully. thereby reducing the burden on the user in typing these *replies*.

3.4.4 User burden reduction

Figures 3.17 shows the user burden, expressed as the average time taken to type a *reply* for the 15 users in the ENRON dataset using a plain keyboard, Swype[116], Google's Smart-Reply system [69] and baseline *Dejavu*. User burden is defined as follows for each of the four media:

- *User Burden with Keyboard* = $\frac{Response\,Length}{Typing\,speed(words\,per\,minute)}$

Table 3.5: Examples of email snippets

Similarity	Email snippets
0.69	**Email:** Carol St. Clair asked me to schedule a meeting regarding the review of pulp and paper's confidentiality agreements. I have tentatively set it for Friday, September 10 at 10 AM. Let me know if this day and time works for you? **Reply:** works fine for me. **Suggestion:** Please put on my calendar
0.68	**Email:** I can't remember if your call is at 9 or 10 Houston time. Please let me know. **Reply:** 9 am Houston **Suggestion:** I am not sure I will be able to (or even should) speak on Friday for our 9 am (Houston) conference

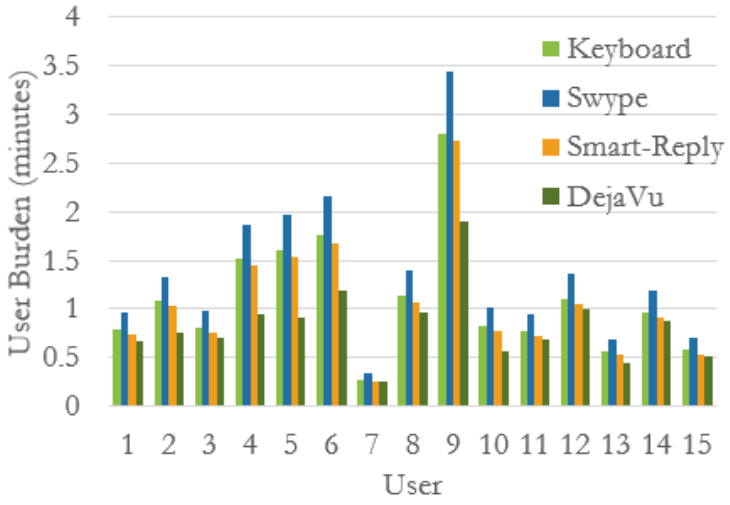


Figure 3.17: Reduction in User Burden

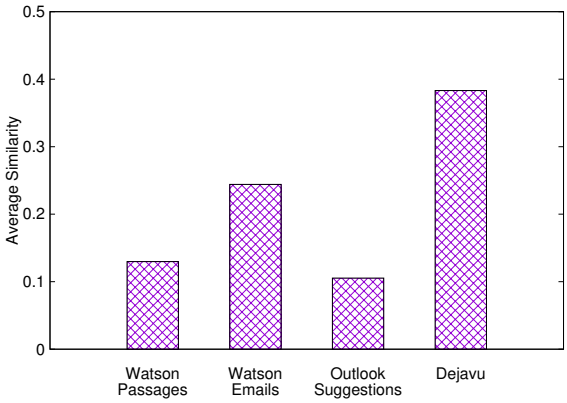


Figure 3.18: Average similarity between suggestions and *reply* for user Causholli-M from the ENRON dataset

- *User Burden with Swype* = $\frac{Response\,Length}{Tracing\,Speed(words\,per\,minute)}$
- *User Burden for Smart-Reply and* Dejavu = $\frac{Response\,Length(1-similarity(suggestions,response))}{Typing\,Speed(words\,per\,minute)}$, if the suggestion was used in creating the response; $\frac{L_r}{(Typing\,Speed(words\,per\,minute)}$ if the suggestion was not used

For all of the users, the burden with *Dejavu* is the lowest to type a *reply*. Also, for these users, the burden with Swype is consistently the highest. In particular, *Dejavu* reduces the user burden by 21% compared to the keyboard, 35% compared to Swype and 16% compared to Smart-Reply. While *Dejavu* and Smart-Reply both suggest replies to ease user burden, the responses suggested by Smart-Reply are non-informational in nature and are short phrases. In contrast, *Dejavu*'s suggestions are longer and informational in nature. Hence, user burden with *Dejavu* is lower than that of Smart-Reply.

3.4.5 Performance Comparison to Related Approaches

In this section, we compare the relevance of *Dejavu*'s *reply* suggestions with an Information Retrieval based related approach (IBM Watson Discovery[62]) and a Smart-Reply system (Outloolk Suggestions [71]). While there are several other systems that can be adapted for automated suggestions in practice, we contend that Watson Discovery and Outlook Suggestions present a representative sample for information retrieval and deep recurrent neural

network based systems, respectively.

To obtain suggestions using IBM Watson Discovery, we upload the body content for all emails for a user into a collection and annotate them with information from email headers like the message ID, subject, senders/receivers, folder (Inbox or Sent), reply content, and the timestamps at which the emails were sent/received. The uploaded content is indexed and stored using proprietary indexing algorithms by Watson. We then query the collection using text from an Inbox email *em* requesting 3 passages $(p1, p2, p3)$ and 3 emails $(e1, e2, e3)$ having a timestamp less than the timestamp of the email. We compute the average similarity between the retrieved passages/emails to $reply$ of *em* as if the passages/emails were suggestions.

For each email in a User's Inbox, Outlook determines if it requires a reply and presents 3 short non-informational phrases as reply suggestions. These phrases are displayed by default at the bottom of the text of the email. As there is no API by which we can obtain these suggestions automatically, we adopt the following procedure to get suggestions. We create two email accounts, one corresponding to the email account of the user ($USER$), and one corresponding to email senders for the user ($SENDER$). We initally preload the $USER$ account's Inbox and Sent folders with all emails in the user's mailbox. We then send each email that has a reply from the $SENDER$ account to the $USER$ account and manually note the three suggestions presented from the Outlook web client on a browser. We then compute the average similarity between these suggestions and the actual $reply$ typed by the user. Note that the average similarity computed this way will be a upper bound on the performance of Outlook suggestions in practice, as we preload the email account with all emails (including the ones for which we obtain the suggestions later) in advance.

Figure 3.18 shows the average similarity for one user 'Causholli M.' from the ENRON dataset, for the suggestions from *Dejavu*, Outlook and Watson (for both passage retrieval and email retrieval). For this user, we can observe that *Dejavu* outperforms all the related approaches. *Dejavu*'s suggestions are 3.6X, 2.95X and 1.56X more similar to the actual

replies than Outlook Suggestions, Watson's passage retrieval and Watson's email retrieval, respectively.

3.4.6 Performance of *Dejavu++*

In this section, we evaluate the impact of the optimizations proposed in Section 3.3, namely, topic filtering, including user feedback, and extending to the global network of mailboxes. We evaluate each of these optimizations individually on the 20 users from the AVOCADO dataset and compare it to the performance of *Dejavu*.

Dejavu++ *with Topic Filtering*

For each user in the dataset, we first sort the emails in the mailbox by their arrival timestamp and use the first 75% of the dataset to populate the Information Database. After that, we compute suggestions using baseline *Dejavu* and *Dejavu++* with topic filtering. We evaluate the $hitrate$ of these suggestions on a testing dataset comprised of the other 25% of the user's mailbox. For the information database containing 75% of the user's mailbox, we extract the topic-word and document-topic probability distributions using the LDA model from Scikit-learn[117]. We set the number of topics K to 20 and the number of labels L to 3. For each entry from the database, we obtain the top 3 topics (using the document-topic probability distribution) and label the entry with these topics. For each email in the testing dataset with a reply, we also obtain the top 3 topics and match the keywords of this email with entries in the database labeled with the 3 topics.

To study the effect of *Dejavu++*'s topic filtering optimization on improving the suggestion computation time, we calculate the average number of entries an email is matched to when computing the suggestions. Figure 3.19 shows the average ratio of entries excluded from the search to the total number of entries in the information database for the 20 users in the AVOCADO dataset. We can observe that topic filtering indeed reduces the complexity of computing suggestions by reducing the search space by 30%, on average

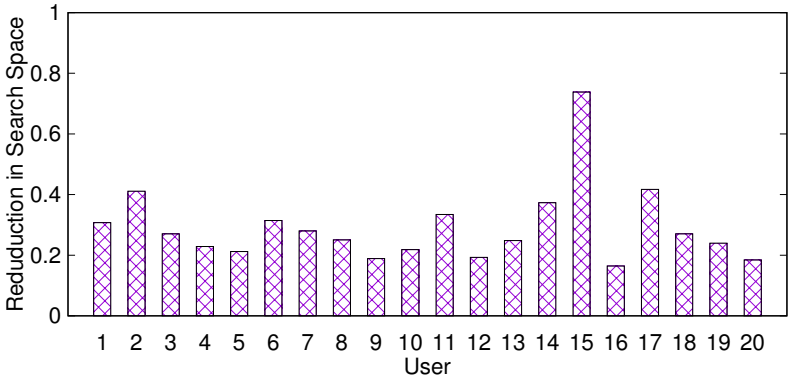


Figure 3.19: Reduction in the complexity of search for suggestions for *Dejavu++* compared to baseline *Dejavu*

across all users. With the reduction in search space due to topic filtering, there is a chance that some entries in the database are wrongly excluded from being a candidate for a suggestion. This happens because we only label the entries with the top $L = 3$ topics. To measure the impact of excluding some entries with topic filtering, we also calculated the $hitrate$ for one suggestion for the AVOCADO dataset and found that the decrease in the $hitrate$ is negligible (only 2.5% of the $hitrate$ without topic filtering). Therefore, the topic filtering optimization of *Dejavu++* is effective in reducing the complexity of suggestion computation without compromising on the performance.

Dejavu++ *with scoping*

Given an email em with a response r_{em} in the testing dataset, we extract the entry em' from the Information Database with the highest similarity to the response r_{em}. We set the predicted length ($L_{em}^{(p)}$), time to respond ($T_{em}^{(p)}$) and sender/receiver email set ($E_{em}^{(p)}$) to the length ($L_{em'}$) of em', the time difference between the timestamp of em' and em ($T_{em'} - T_{em}$) and the set of senders and receivers of em' ($E_{em'}$). We set the margins for length (ε_L), time to respond (ε_T) and the sender/receiver email set (ε_E) to be 20 words, 1 week and 1 email address, respectively. Note that even though we do not predict the length, time to respond and email-set and choose an ideal fixed value for these parameters, we choose high

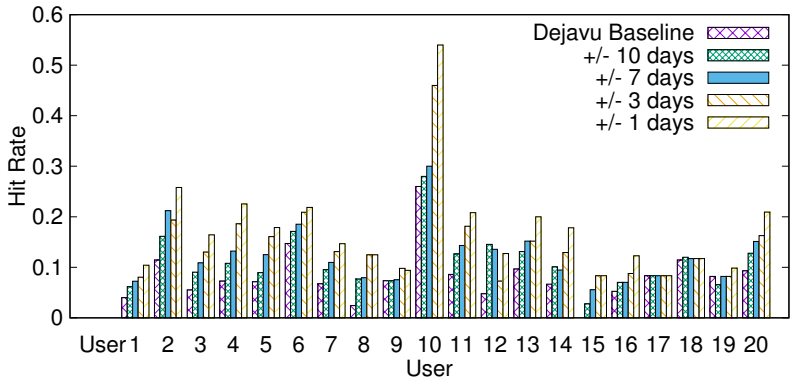


Figure 3.20: Hitrates for different values of ε_T for the AVOCADO dataset

margins for the length and time and a low margin for the email-set to account for any errors in prediction. The prediction of these parameters is beyond the scope of this work.

Figure 3.20 shows the performance of *Dejavu++* when only the expected time to respond is considered when matching the email *em* with the entries of the Information Database. We set the expected time to respond to be equal to $(T_{em'} - T_{em})$, where T_{em} is the timestamp of the email and $T_{em'}$ is the time stamp of the best possible match (em') to the response of *em*. We calculate the hitrate as we vary the value of ε_T from 1 day to 10 days. We can observe that as the margin ε_T increases, the hitrate also increases for most users. The average hitrate across 20 users is 0.17, 0.14, 0.12 and 0.11 for ε_T values of 1day, 3 days, 7 days and 10 days, respectively. The expected time filter optimization improves the hitrate by 112.5%, 75%, 50% and 37.5% respectively for ε_T values of 1 day, 3 days, 7 days and 10 days, respectively.

Figure 3.21 shows the hitrates of *Dejavu++* for the 20 users in the AVOCADO dataset when the matching algorithm filters only by the expected length of the suggestion. We set the expected length of the suggestion to length of the best possible match of the response and vary the margin ε_L from 0 to 50 words. We can observe that as the margin increases, the hitrate decreases. The average hitrates (across 20 users) are 0.27, 0.17, 0.16, 0.15, 0.15 and 0.13, respectively, for ε_L values of 0, 10, 20, 30, 40 and 50, respectively. As the ε_L value increases from 0 to 50, the improvement in the hitrate (compared to baseline *Dejavu*)

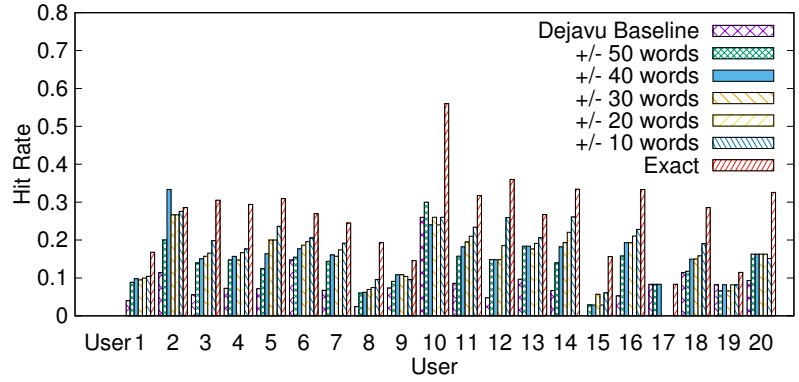


Figure 3.21: Hitrates for different values of ε_L for the AVOCADO dataset

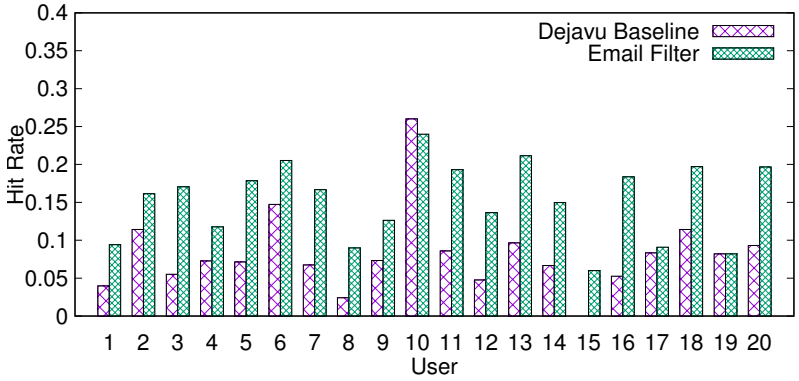


Figure 3.22: Hitrates with expected sender/reciever email-set filter for AVOCADO dataset

decreases from 237.5% to 62.5%.

Figure 3.22 shows the hitrates of *Dejavu++* for the AVOCADO dataset when only the expected senders/receivers are considered when computing suggestions. We match each email *em* in the testing dataset with a response to every entry from the Information Database with atleast one sender/receiver in common with the best possible match (em') of the response to the email *em*. We can observe that filtering on expected sender/receivers improves the hitrate by 87.5%.

Dejavu++ *with User Feedback*

In this section, we evaluate the performance of *Dejavu++* on the 20 users in the AVOCADO dataset with user feedback optimization. Unless otherwise mentioned, we set the hitrate

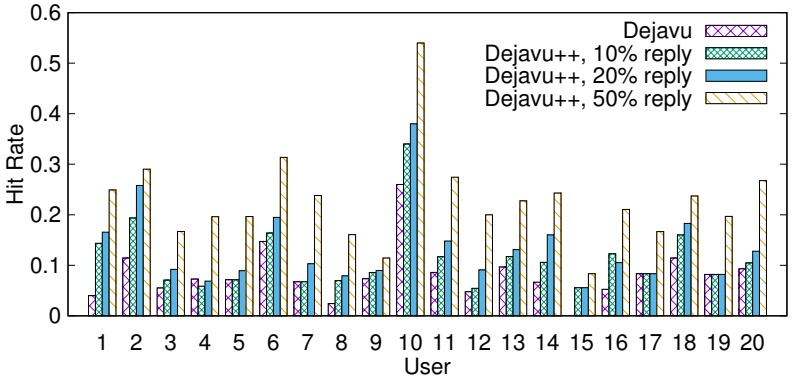


Figure 3.23: Performance of *Dejavu++* with user feedback

threshold τ to 0.6, and the number of suggestions to 1.

Figure 3.23 shows the performance of *Dejavu++* for the scenario where the user looks at the initial suggestions, uses them if relevant, or types the *reply* from scratch otherwise. For each email *em*, we classify the suggested response $s(em)$ as *relevant*, if the degree of match (as defined in Equation 3.1) between the response and the suggestion is greater than 0.6. Initially, the keywords of the email *em* are matched with the information database to compute suggestions $s(em)$. If $s(em)$ is irrelevant to *em*, we recompute the suggestions after setting the relevancy flag $irel_{s(em)}$ to 0, thereby marking the old suggestion as irrelevant in the database. We then use keywords from the partially typed reply to find a new suggestion $s_{new}(em)$ for the email. The $hitrate$ for *Dejavu++* is the ratio of emails with similarity with $s_{new}(em)$ is greater than the threshold τ. The $hitrates$ for *Dejavu* and *Dejavu++* with suggestions recomputed when the user has typed 10%, 20% and 50% of the actual reply are shown in Figure 3.23. We can observe that as the percentage of partial reply available to *Dejavu++* at the time of refreshing the suggestions increases from 10% to 50%, the $hitrate$ improves. Specifically, compared to baseline *Dejavu*, the inclusion of user feedback improves the average $hitrate$ across the 20 users by 37.5%, 62.5%, and 187.5%, respectively, when at 10%, 20% and 50% of the actual reply, respectively. While the $hitrate$ increases for all users with user feedback, the magnitude of improvement varies. For some users (e.g., user 18), the improvement in $hitrate$ is close to zero, even with the inclusion

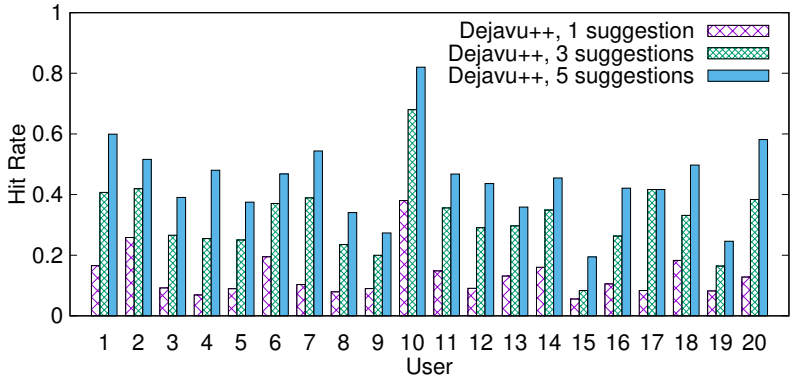


Figure 3.24: Impact of the number of suggestions on *hitrate* for *Dejavu++* on the AVOCADO dataset

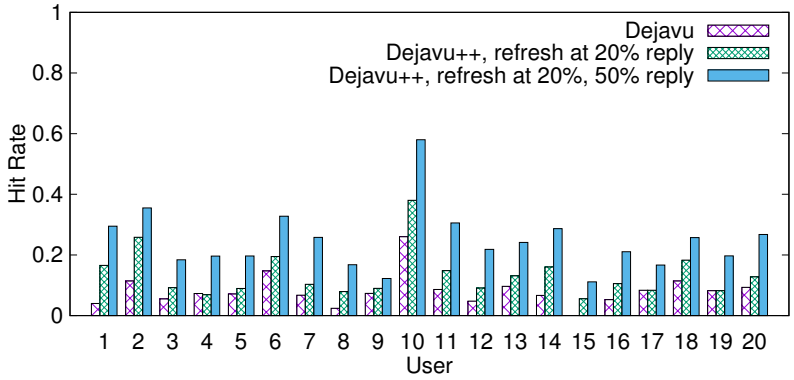


Figure 3.25: Impact of number of times suggestions are refreshed on the *hitrate* for *Dejavu++* on the AVOCADO dataset

of 20% of the reply. For these users, the majority of keywords with high IDF scores occur in the later portion of the reply. Therefore the suggestions computed with the first few keywords of the reply may not have the higher weight keywords of the reply within them.

Figure 3.24 shows the *hitrate* for *Dejavu* and *Dejavu++* for 1, 3, and 5 suggestions, for a scenario when the user does not find the initial suggestions relevant and has already typed 20% of the response. We can observe that as the number of suggestions increases, the hitrate improves. On average, the *hitrate* improves by 1.46X for three suggestions and 2.38X for five suggestions. This can be attributed to the increase in the information available from suggestions to include in responses.

Figure 3.25 shows the impact of *hitrate* on the number of times suggestions are recom-

puted in *Dejavu++*. For this experiment, we assume a scenario where the user looks at the suggestions (i) initially (shown as *Dejavu* on the figure), (ii) after typing 20% of the response (shown as *Dejavu++*, refresh at 20% reply on the figure), and (iii) after typing 50% of the response (shown as *Dejavu++*, refresh at 20% and 50% reply in the figure). With an increase in the number of times suggestions are refreshed, the *hitrate* improves. Specifically, compared to baseline *Dejavu*, refreshing suggestions at 20% reply improves the *hitrate* by 62.5%. Subsequent refresh when the user has typed 50% reply further improves the *hitrate* by 212% compared to baseline *Dejavu*. Note that the *hitrate* for a refresh at 20% reply and 50% reply is higher than the *hitrate* for refresh only at 50% (see Figure 3.23. With more refreshes, *Dejavu++* excludes more irrelevant entries from being considered as suggestions.

Dejavu++ *with the global network of mailboxes*

Figure 3.26 shows the *hitrate* of 20 users of the AVOCADO dataset when only the global network mailbox optimization is turned on. On average, this optimization results in an average *hitrate* of 7%. The average *hitrate* of *Dejavu++* with global network optimization is lower compared to baseline *Dejavu*. We can observe that optimization does not benefit all users and reduces the *hitrate* (compared to baseline *Dejavu*) for some users. For this subset of users, even though an email *em* may be more similar to entries from other users' information databases, than the entries within the user's database, the actual response may not be close to these entries. Therefore, with the global network mailbox optimization, *Dejavu++* can provide relevant suggestions to only a fraction of emails.

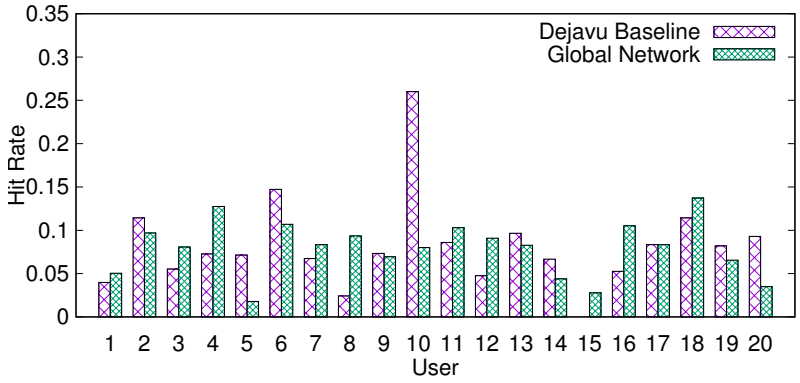


Figure 3.26: Hitrate with the inclusion of global network of mailboxes for AVOCADO dataset

CHAPTER 4

TASKR: FAST AND EASY MOBILIZATION OF SPOT TASKS IN ENTERPRISE WEB APPLICATION

4.1 Introduction

The adoption of mobile devices, and in particular smartphones, has grown steadily over the last decade. Fifty-one percent of enterprise workers today use mandated apps for their business on their phones [118]. Seventy-seven percent of the workers rely on their personal smartphones to perform their work [119]. One of the key drivers of the adoption and use of smartphones is the self-perceived increase in productivity. Employees self-reported getting an hour of time back by relying on smartphone apps for their work. Intriguingly, employees were relying as much on company-issued mobile apps as they were on *bring your own application* apps [120].

Consider an enterprise worker, Alice, who is a field salesperson. An average enterprise runs 400+ applications for its business operations. Alice is likely to interact with many of these applications, with examples ranging from Oracle HR, SAP ERP, Microsoft Sharepoint, and Salesforce CRM. If Alice desires to do some of her Salesforce tasks on her smartphone when she is away from her desk, she currently has to be dependent on either Salesforce releasing a mobile app or her employer building a custom mobile app that taps into the Salesforce APIs. In both cases, not only does the mobile app for Salesforce need to exist, but her specific task also has to make the cut through the de-featurization process necessary for mobilization, and has to be achievable with minimal burden within the design of the mobile app.

Interestingly, in spite of the increasing adoption of mobility in enterprises, over eighty percent of enterprise mobile apps are abandoned after the first use [121]. In this context,

we identify a category of tasks called *Spot Tasks* within enterprise web applications, and present a strategy wherein Alice can perform the desired mobilization *herself* and *without requiring any support from either the application vendor or the enterprise*. We define spot tasks as tasks that can be accomplished by the users interacting substantively with the desktop application *only on a single page*. The interaction on that page could be in the form of read, act, and navigate actions. Also, that specific page could be arbitrarily anywhere within the applications navigation tree. While we relax these definitions in subtle ways later in the chapter, we also show how even such a constrained definition can support a wide variety of enterprise task profiles.

For example, consider a purchase approval task on a typical SAP SRM (supplier relationship management) application. This could require the user to login and authenticate herself, navigate to My Work, navigate to Purchase Management, navigate to Requisition Approvals, see a list of approval requests, check on those requests that need to be approved, click on the Approve button, and finally logout of the application. In this example, the first sequence of pages visited is for navigational purposes while the purchase request review and approval are done on a single page. Thus, we categorize such a task as a spot task.

Spot tasks are limited in capabilities, but have several critical advantages that make them an interesting candidate for a mobilization strategy. We present a mobilization solution called *Taskr* to mobilize spot tasks that exploits these advantages and delivers the following properties:

- Configuration by doing: *Taskr* allows the user to perform the mobilization herself regardless of the users technical skills. All *Taskr* requires for the mobilization of a spot task is for the user to be able to *perform the spot task* on the desktop application;
- Programmatic APIfication: Once the user configures what needs to be mobilized, *Taskr* programmatically creates the necessary APIs using purely a front-end strategy [1] that requires no access to source code from the application vendor, or even special provisions

[1] We elaborate later in the chapter, but at a high level this involves relying on a remote-computing based approach to create the APIs.

by the enterprise;

- Guard-rails: Since SpotTasks are restricted to purely navigational actions till the final interaction page is reached, and no further navigations are allowed, the workflow is by design *simple*. This allows Taskr to be avoid configuration-time versus run-time errors.
- Flexible mobile delivery: Since spot tasks are restricted to a single interaction page, and *Taskr* further imposes limits on the amount of content and actions mobilized on the interaction page, it allows for flexible delivery mechanisms on the smartphone. *Taskr* allows the user to consume the mobilized tasks through Twitter (direct messaging), Email, and a Native Mobile App.

We implement *Taskr* on an AWS backend and an Android frontend, and conduct preliminary user experiments to evaluate its performance. The results are promising and show that not only does *Taskr* reduce the actions required to complete tasks (by over 35%) but also that users are more satisfied completing spot tasks with *Taskr* compared to the desktop or the mobile browser (by over $7x$). The rest of the chapter is organized as follows: We define spot tasks in Section 4.2 and introduce *Taskr*'s design in Section 4.3. We then evaluate it in Section 4.3.2. Finally, we discuss some issues with *Taskr* in Section 5.6

4.2 Mobilization and Spot Tasks

4.2.1 Mobilization and Defeaturization

Application mobilization is the process of adapting applications originally built for desktop environments for use on mobile devices. Enterprises typically mobilize applications by (i) building native apps from scratch [122], (ii) using vendor applications [5], (iii) using mobile-specific backend libraries (mBaaS) [123] and (iv) with custom app development platforms (MEAPs) [73].

Enterprise desktop applications are complex and allow a wide variety of business functions. These applications support a large number of workflows - wherein each workflow

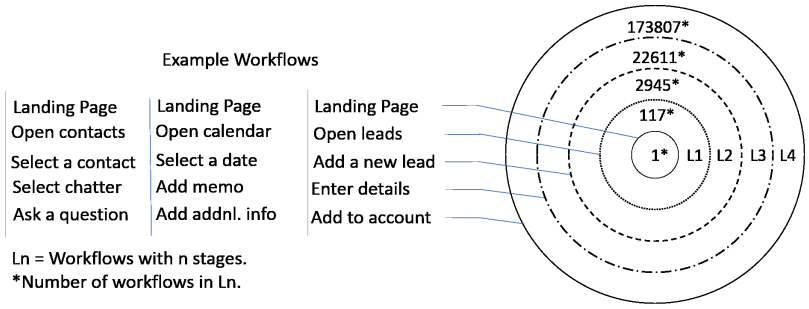


Figure 4.1: Complexity of the Salesforce desktop application

represents a goal-oriented series of actions taken by the user[2]. Considering the constraints of the smartphone, it is not feasible for a mobile app to support all the desktop application workflows. Therefore, the desktop application has to be *defeaturized* before it can be mobilized.

To quantify the complexity of enterprise applications, we recursively crawled the Salesforce CRM web application and observed that there are over 180K workflows for just 4 levels. In Figure 4.1, each circle shows the number of possible workflows (along with examples) that a user can perform starting from the landing page. We observed that there are over 180K workflows for just 4 levels [3]. It is impossible for a mobile app to support all of these workflows, and provide a good user experience at the same time. On the other hand, in Salesforce1 mobile app (the mobile version of Salesforce CRM), there are only 48 navigational workflows at the first level (as opposed to 117 in the desktop version). When a complex enterprise application such as Salesforce CRM with thousands of workflows needs to be mobilized, one of the key problems to be addressed is what workflows are made available on the mobile app. This critical step in application mobilization is called **defeaturization** - wherein the number of features exposed on the mobile app is reduced to a fraction of what the original desktop version supports. The defeaturization decision today is taken either by the enterprise (if custom mobile app development is done), or by the application vendor (if it is an off the shelf or a SaaS app).

[2]For example, Salesforce has about 180K navigational workflows with just 4 navigational steps

[3]Note that in this example, we only consider workflows involving links that lead to a new URL, and therefore Figure 4.1 is only a subset of the total number of possible workflows (with 4 stages).

The granularity of defeaturization i.e. the number of features retained in the mobile app, typically lies in the following spectrum:

- *The entire web application along with all the features are retained in the mobile app.* Considering the desktop application as a large collection of pages, the structure of the pages within the application is largely maintained. This granularity is chosen when all the features within the application are heavily used;
- *A subset of features from the original application, carefully chosen either by the enterprise or the vendor, are mobilized.* Given that enterprise web applications are complex, and that the mobile device cannot support all the features, a subset of features from the original application is carefully chosen either by the vendor or the enterprise itself. The features to be mobilized are chosen based on how heavily they are used and the requirements of the user's job functions. Here, a subset of the pages of the original application, and a subset of the features on those pages are mobilized. With this strategy, the structure of the pages among the application is largely maintained, while reducing the number of features on any given page;
- *A mobile-first approach that uses APIs provided by the application to build the mobile app ground up.* Many applications expose some of their features through APIs that can be leveraged for different purposes. This approach can only mobilize those features that have been exposed as APIs; This is a mobile-first approach where in the mobile app is built ground up using these APIs, independent of the structure of the original application; The structure of the resultant mobile app need not be the similar as the original application.
- *A sequence of features that constitute different steps of a single workflow are mobilized*. In this case, once the user starts the workflow on the mobile device, only the features relating to this workflow are presented, thereby decreasing the effort of finding a feature.

4.2.2 Spot Tasks

In this chapter, we identify another potential defeaturization granularity - *Spot Tasks*. A spot task is a simple linear workflow within an enterprise application where-in all the user interactions are only performed on one page of the application. These user interactions constitute the last stage of the workflow. However, this page can be buried deep within the complex application and the navigational effort required to reach that particular page may be high. Spot tasks can also be paused and resumed at a later point of time as long as the current session is maintained on the browser[4].

UI elements within an application page can be classified as: (i) READ: elements that carry content that is only consumed by the user (e.g., text content of an article); (ii) ACT: elements through which the user writes some parameters in the web application (e.g., text boxes to enter values, dropdown lists, etc.); and (iii) NAV: elements that progress the workflow to the next stage (e.g., links, submit buttons, etc.); The next stage of a workflow can depend on the user's actions on the ACT and NAV elements of the current stage.

For a spot task, each stage of the workflow, except the last stage, has only one NAV element and the final stage of the workflow can have READ/ACT/NAV elements. In other words, if the presence of READ, ACT, and NAV elements in a stage is denoted as R, A, and N, respectively, and the end of a stage is denoted as X, the spot task can be described using a regular expression as follows:

$$ST = [NX] * R?A?N?X$$

Note that even such a constraining definition of spot tasks still covers a substantial number of workflows within enterprise applications. We identify 45 spot tasks within 9 enterprise applications in Section 3.3. For example, checking the revenue on Salesforce, adding a vendor on Quickbooks, and viewing the available vacation days on Oracle Peoplesoft are

[4]Note that the pause and resume needs to happen on the same device and the same browser instance.

all spot tasks (assuming the user is logged in).

Spot task variations: In this chapter, we further expand the definition of spot tasks to also account for workflows with fixed (non-variable) inputs along all the stages except the last stage. In other words, for every instance the spot task is executed, every stage except the last stage always has a fixed value for ACT elements and a fixed NAV element. The non-variable inputs allow for the hard coding of the ACT actions needed to reach the final screen where the user actions are performed. If the presence of fixed ACT values is indicated by F, the spot tasks can be expressed as - *[F?NX]*R?A?N?X*. If the user is required to enter a username and password before executing a workflow, then all of the previous examples are still spot tasks under this definition (username and password are fixed values) [5]. The requirement of non-variable inputs except at the last stage of the workflow is further relaxed for login parameters like username, password, and domain. The values can change with different runs of the spot tasks, but the result of the workflow after the variable-parameter login stage needs to be fixed across each run of the spot task.

Mobilizing Spot Tasks: The simplicity of spot tasks empowers the users in a significant fashion wherein the users can drive the mobilization efforts themselves, regardless of their individual skills. The granularity at which mobilization has traditionally been performed necessitate the enterprises to invest significant resources and employ developers with specialized skill sets. Further, the resultant enterprise mobile apps are constructed in a one-size-fits-all fashion and are unlikely to address the needs of all required business functions performed by the entire user base within an enterprise. Thus, for many users, there will exist workflows that the resultant mobile app (i) will not support at all; or (ii) have a considerably increased task burden to perform.

However, if there exists a mobilization solution that the users themselves rely on to create an app that is custom built for their workflows, these issues could indeed be addressed. The challenge though is how to enable such configuration of the mobile app regardless of

[5] We provide more examples of spot tasks in Section 4.3.2

the skills possessed by the user, and also, how the resultant mobile app can be made user-friendly. In this work, the only skill that we assume from the user is the ability to *perform the workflows (to be mobilized) on the desktop.* Since the user performs the workflows on the desktop anyway, this is a valid assumption.

The simplicity of the spot tasks allows for the design of such a mobilization solution to be possible. Since the spot tasks have a limited number of UI elements from within only one screen of the application, easy configuration of the apps (and the layouts) can be achieved, without requiring the user to have coding and design skills. Also, the linear non-parametric nature of spot tasks allows for the creation of robust mobile apps. Since the value of ACT and NAV elements are fixed for spot tasks, the sequence of stages in the workflow will always be the same. This eliminates the need for the user to anticipate any branches that may depend on the value of ACT/NAV elements and configure them. For workflows with variable ACT elements, it is possible that the value of ACT element entered influences the next stage of the workflow. This can result in the complex branched workflows. It is not feasible to assume that the user has skills to anticipate all such branches and configure them. If the user fails to configure a branch, it can result in errors and unfinished workflows. The inherent simplicity of spot tasks makes them ideal candidates for mobilization. Furthermore, even if these tasks are already mobilized under other granularities, the users still might have to experience navigational burden just to perform these simple tasks.

Scope and Goals: The scope of our work is limited to the mobilization of spot tasks within enterprise web applications. We primarily consider HTML/JS compatible web applications due to their dominance [124]. However, the design principles are applicable to other platforms. The solution needs to support all major smartphone OSs (Android, iOS, Microsoft). The solution also needs to be usable by all users regardless of their skills.

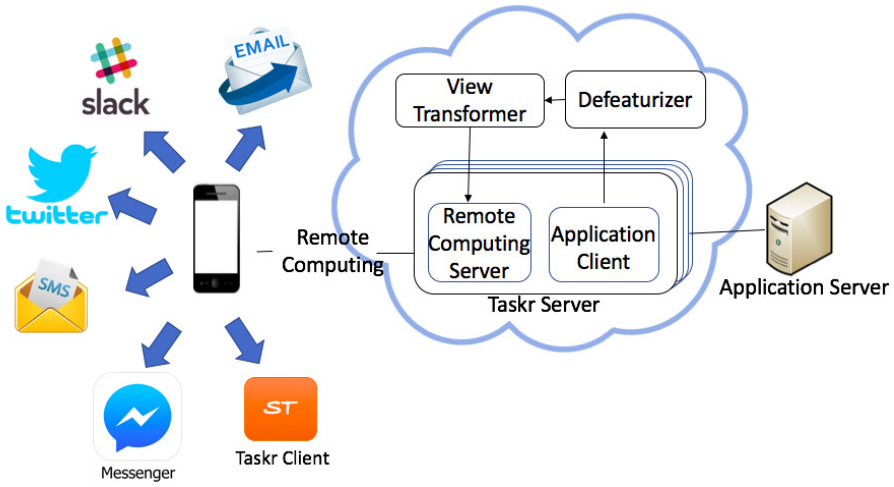


Figure 4.2: *Taskr* Architecture

4.3 Taskr: A Do-it-Yourself Approach to Spot Task Mobilization

In this section, we present *Taskr*, a framework that allows for mobilization of spot tasks within enterprise applications by all users. The *Taskr* infrastructure consists of three components - *Taskr-recorder*, *Taskr-server* and *Taskr-client* (Figure 1).

- The *Taskr-server* is hosted on a cloud platform. The infrastructure has a control plane to allow DIY configuration, service requests, host enterprise application clients, maintain transformation rules and logging.
- When the enterprise wants to allow DIY mobilization for a particular application, it hosts the corresponding application client (for web applications, this would mean a browser pointing to the appropriate URL) on the infrastructure.
- When a user wants to mobilize her workflows, she uses the *Taskr-recorder* configuration tool to configure the mobile app simply by *performing the workflow that needs to be mobilized.*
- The infrastructure generates a *Taskr-client* mobile app (.ipa, .apk, and URL) for the user to download and install onto her smartphone. In addition to this app, the user can also use Twitter or email to perform her workflows. The mobile app is simply a remote viewer with the appropriate application and user configuration locator embedded in it.

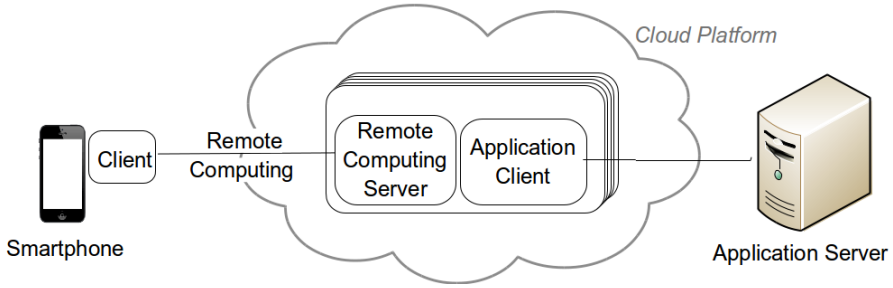


Figure 4.3: Remote Computing

- In addition to generating the app, *Taskr-server* also maintains a communication handle for the user corresponding to the different usage modalities. When the user launches the *Taskr-client* app, a computing slice is set-up on the fly to service that specific user session. The slice automatically loads the corresponding desktop application and user configuration.
- The infrastructure delivers the mobile view as configured to the smartphone. The user interacts with the *Taskr-client* app, and the actions are shipped to the cloud infrastructure where they are performed on the desktop client. In addition to the mobile app, the user can also start the spot tasks by sending a command to the *Taskr* over Email, Twitter, SMS, Slack, etc. The server replies to the user with any configured READ elements and asks the user to send the values of the configured ACT elements. The user can then reply to this message with the ACT values.
- Any changes to the client view either based on server pulls or pushes is appropriately transformed and the corresponding mobile friendly view delivered to the mobile app.

We now delve into the key design elements of *Taskr*.

4.3.1 Key Design Elements

Remote Computing with Refactoring:

Taskr uses remote computing [125, 126] to mobilize applications while requiring no development and minimal deployment effort from the enterprise or the end-user. Figure 4.3

shows the architecture of remote computing based framework for mobilization for a target application with a client-server architecture. To mobilize any given application, enterprises can host a remote computing server and the application client on a Virtual Machine in the cloud. The application client's view is then streamed to the remote computing client on the user's smartphone. The user interacts with the application locally on her smartphone. With remote computing, the users can access all the rich features of the enterprise applications on their smartphone.

Mobilization through remote computing can be achieved quickly without requiring significant development and deployment effort. It is indeed an interesting candidate to solve the mobilization problem as users could conceivably be provided with a simple framework that they could configure to mobilize applications. However, the key limitation of remote computing is that it does not allow for any meaningful defeaturization. The entire application is streamed to the smartphone as-is. Given the complexity of the application and the fact that any particular user is only interested in a subset of the features provided by the application, this method is very likely to increase the usage burden. The user will have to navigate the complex application on a much smaller screen than what the application was originally developed for. If the entire application view is presented at once to the smartphone user, the UI elements appear very tiny requiring significant zooming effort from the user. On the other hand, if the size of the desktop view is maintained, significant scrolling effort will be required from the user.

While thin-client remote computing offers a easy way to mobilize an application while retaining all the features of the application, the result is not usable at all on a smartphone. It is clear that the remote application view presented has to be optimized for the client device. However, unlike traditional remote computing, *Taskr* optimizes the remote view for the client device through *Application Refactoring*, wherein the desktop application UI is dynamically transformed into an appropriate UI for the smartphone. Refactoring restructures the view for the target platform without changing the underlying application behavior via

two steps - (i) reducing the number of features available (*Defeaturization*) and (ii) optimizing the application view (*Transformation*). This view is dynamically generated as the user is using the application. The original application view is defeaturized and transformed to suit a corresponding input modality chosen by the user. For the native mobile app modality, this means the UI elements of the original view are transformed into a smartphone native view. On the other hand, for text based input modalities like email, twitter, SMS, etc., the UI elements are transformed into their corresponding text versions. Any actions on these transformed UI elements are sent to the application client and virtually executed on it. While the user can simply perform these actions using a mobile app modality, she can describe these actions over the text based modalities. For example, to check the account balance, the user can simply send (using an email/tweet/SMS/etc.) a command with username and password values appended, asking for the account balance to the server. The server can execute the workflow with the username and password values and reply back to the user with the account balance.

The benefit of this approach stems from the fact that the UI elements of the desktop application can be selectively chosen and transformed into highly optimized versions for usability on the smartphone. The dynamic transformation also allows for the user to perform the tasks using several input modalities such as tweets, email, SMS, Slack, messenger, in addition to smartphone native app. By using application refactoring as a basis for mobilization, *Taskr* marries the benefits of no coding, scripting, or development of remote computing to the usability of true native apps. Furthermore, if the user is allowed to configure the defeaturization and transformation process while maintaining the usability of the resultant mobile app, refactoring presents a powerful paradigm through which enterprises can solve the mobilization problem for spot tasks.

Do-It-Yourself Configuration:

Users of an enterprise application best know what features are required to be present in the mobile app, in order to perform their job functions easily. *Taskr* leverages this fact and allows the users to mobilize the necessary features themselves while simultaneously maintaining the usability of the resultant mobile app. *Taskr* only requires the users to know how to perform the workflow on the desktop application.

For configuration, the users are only required to perform the workflows on the Desktop application in the presence of a configuration tool - *Taskr-recorder*. This tool observes the user's interactions with the application to know what UI elements are necessary for the completion of the task and defeaturizes the application to include only these elements. In addition to performing automatic defeaturization, the tool also allows the users to fine-tune the configuration through an intuitive user interface. Also, the transformations of these UI elements (into optimized smartphone versions) are computed simultaneously and presented to the user. This transparency not only allows the user to verify the mobilization results, but also to modify when needed.

Flexible Mobile Delivery:

The result of the configuration process is a mobile app through which the users can view all their spot tasks and execute them. Note that a key goal of *Taskr* is to reduce the task burden of performing the tasks for all users irrespective of their skill levels. Therefore, *Taskr* does not restrict the users to use a mobile app to execute the tasks and extends the user interface to include other usage modalities. Smartphone users use certain apps extensively throughout their day (e.g., Twitter, SMS, Email, Slack, Messenger, etc.). *Taskr* leverages the users' familiarity with these modalities and allows them to execute their tasks within them. This saves the user the burden of learning to use the interface of a new mobile app - *Taskr* client.

Taskr transforms the UI of the desktop application to suit these usage modalities i.e. smartphone native UI for the *Taskr-client* app and text blurbs for the other modalities. For the *Taskr-client* app, the UI elements are transformed into a smartphone native elements. On the other hand, for the other text based modalities like email, twitter, SMS, etc., the UI elements are transformed into their corresponding text versions. Any actions on these transformed UI elements are sent to the application client and virtually executed on it.

Single Screen Transaction:

The ideal candidates for DIY mobilization are the workflows that can be performed easily with the limited screen real estate of a smartphone. The workflows should not only require little user interaction but also be simple enough to be configurable by users of all skill ranges. Therefore, in order to maintain usability while at the same time requiring minimal intervention from the user, *Taskr* restricts the users to configure only a limited number of UI elements within one spot task. In this work, we set the limit to 140 characters each for the total character count of READ elements and the labels of ACT elements[6].

4.3.2 Challenges and Design Choices

In this section we present *Taskr* Do-It-Yourself application mobilization solution through a series of discussions on design challenges and choices.

How is the configuration done?

If mobilization is done at the granularity of spot tasks, a related challenge is how the configuration is done to indicate what specific spot tasks need to be mobilized. Recall that the fundamental goal is to make the configuration process accessible to the layman user, regardless of skills. Also, since a key goal is to reduce the task burden for a particular user, the tasks to mobilize should ideally be learned from the usage patterns of the desktop

[6]This restriction is arbitrary and is imposed to allow all transactions to fit within a few text messages

application. The output of this process should identify the workflows that can be executed in stand-alone fashion i.e. the completion of the workflow should only depend on user actions from that workflow, and should not depend on any other user actions. For example, consider the case of creating an invoice in a supply chain management application - (A) the user logs in, (B) clicks the invoice link, (C) fills in the necessary fields for billing and saves. The workflow learned should contain all three steps. A workflow involving just steps B and C is not a stand-alone workflow as step B is dependent on step A (there is no invoice link unless the user logs in).

One strategy in line with these goals is to automatically extract the workflows from usage-patterns. However, this process is not straight-forward and can lead to irrelevant/incomplete workflows. A lot of applications remember the credentials of the user and automatically login that user (e.g. Single Sign On feature). Also, a set of applications may use a Single Sign On (SSO) feature, wherein the user needs to sign in only once to access any of the applications in the set. This leads to cases wherein the observations alone cannot capture all the stages in a workflow as-is. Also, through passive observation, only the UI elements that the user explicitly acted upon can be identified and not the UI elements read by the user. On the other hand, the user could be directed to write a script describing all the steps of a workflow. However, this requires significant effort from the user and not all users have the skillset to script the workflows.

Therefore, *Taskr* uses an approach that enlists the help of a user, without requiring the user to configure each and every detail of the workflow. The user configures a spot task by simply performing that particular task in the presence of *Taskr-recorder*. For all the stages except the last stage of the task, the tool automatically tracks the UI elements that are acted upon by the user and records the action parameters - ACT elements, their values and NAV elements. For the last stage, the tool has an interface through which users can select any elements that may have been missed and assign a category to them - READ/ACT/NAV. As the user is selecting the elements, the tool records the number of

Goals:

Cannot assume users with coding and scripting skills. Users only know how to perform the workflow on the Desktop client

Design Elements: *Users configure mobilization by performing the workflow on their Desktop with a Recorder*

- Automatic workflow configuration by performing
- Workflow recording through a combination of active listening and user feedback

Pseudocode:

```
configureMobilization ( ) :
    Recorder.record ( )
    // Execute the workflow
    Recorder.stop ( )
Recorder.record ( ) :
    WHILE (NOT stopped) :
        IF detected (action) :
            sendToServer(action, action.element)
        IF detected (config) :
            sendToServer(config.type, config.element)
```

Systems Considerations:

- *Recorder* registers DOM action listeners on UI elements
- The *Recorder* is triggered to send *element* and *action* parameters to *Taskr Server* when it observes an *action* on the *element*
- *Recorder* also allows the user to explicitly select UI elements to be mobilized

Figure 4.4: Overview of configuration with *Taskr*

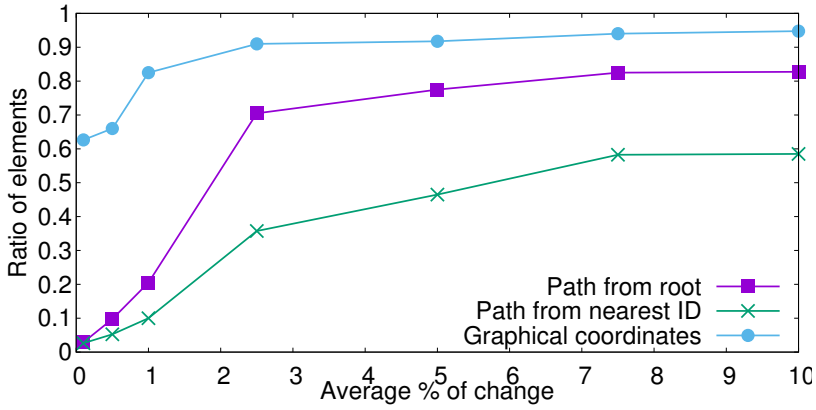


Figure 4.5: Performance of different fingerprint candidates

characters of READ elements and the labels of ACT elements. Once the total number of characters in each category reaches the limit defined in Section 3.3.1, the user is notified and a further selection of elements is disabled. By allowing users themselves to configure the workflows in this simple and intuitive fashion, the needs of that user can be best served. Figure 4.4 shows an overview of configuration with *Taskr*.

How are UI elements identified reliably?

If the actions performed by the user on the refactored view have to be correctly executed by the server, the UI elements involved in a workflow need to be reliably identified among all the other elements in that application, even when the application changes. Otherwise,

the user actions can be incorrectly executed on some other elements. Identification of UI element involves extracting a set of parameters (say, the *fingerprint*) unique to that element in the entire application view. We call this unique identity the *fingerprint* of that element. Given that the web applications are increasingly dynamic today with changing views not only due to the developer modifying the application but also due to end-user actions manipulating data. In such a scenario, the fingerprint of the UI element has to be robust to most changes in the application view.

An obvious choice for an element's identity are its coordinates on the application screen. However, these visual features are not robust. Minor changes (for eg., modifying text of an article) can easily break not only the fingerprint of an element but also the fingerprints of other elements surrounding it. Also, the change in graphical coordinates of one element are trickled down to many subsequent elements, affecting the accuracy of their fingerprints; For example, if the user modifies a wiki page, the graphical coordinates and size of the text of that wiki page change, thereby breaking the fingerprint of not just the changed element, but also all other surrounding elements; Therefore, graphical features are not reliable candidates for identity.

All pages in a web application are structured as a tree, called a Document Object Model (DOM). Each HTML tag in the application is a node in the DOM tree which is rooted at the $<HTML>$ tag. The position of a UI element from the root of the DOM tree is one candidate for a fingerprint. However, it is susceptible to failure due to changes like insertions/deletions/migrations of elements along the path from the anchor to the element. Given the dynamic nature of web applications, it is very likely for these changes to happen, thereby breaking the robustness of the fingerprint. This fingerprint can be further enhanced by considering the path from an anchor element in the DOM, which is closer to the element (to be fingerprinted) instead of the path from the root. This can reduce the probability of layout changes breaking the fingerprints. One candidate for this anchor could be the nearest ancestor on the DOM with an HTML attribute *id*, or the root of the DOM tree when no

ancestor with *id* is found. This anchor element can be uniquely identified on the page by its HTML *id* [7]. This fingerprint is susceptible to changes in the DOM path between the anchor element and the UI element. Furthermore, in some cases, even the *id* attribute is not consistent across different instances of the page. ExtJS [127] is a popular js library that generates a new *id* for every instance of an element.

To verify the robustness of these different fingerprint candidates, we artificially introduced layout/attribute changes into randomly selected nodes of the DOM tree of a Learning Management System application - Sakai [128]. At the beginning of each experiment, we randomly selected 10% of the elements in the DOM tree and extracted their fingerprint. At every iteration (for 5 iterations), with a certain probability, each element undergoes either a layout change or a tag attribute change. Figure 4.5 shows the ratio of elements that are correctly identified after 5 rounds of changes to the DOM tree vs. the % of nodes changing in the DOM tree. We observed that even with 1% of nodes changing, 10% of nodes are incorrectly identified with Path from nearest element with *id* and 82% nodes are incorrectly identified with graphical coordinates. This shows that none of the features discussed so far are ideal candidates for fingerprints.

Therefore *Taskr*, adds resiliency to the fingerprint by using the position of the elements relative to several other elements in the page to calculate an element's fingerprint. More details on the fingerprint are available in chapter 5. Figure 4.6 shows an overview of how *Trackr* is used in *Taskr*.

How is data extraction done?

Once the UI elements are identified, extracting the (i) nature of the UI element (e.g., textbox/button etc.) and any (ii) associated context (e.g., label) is crucial so that the user can understand and execute its function on the mobile device as intended. For example, Bootstrap, one of the most popular UI frameworks, allows developers to integrate complex

[7] Note that the *id* attribute itself cannot be used as a fingerprint because it is not necessary for all elements in the DOM tree to have an *id*

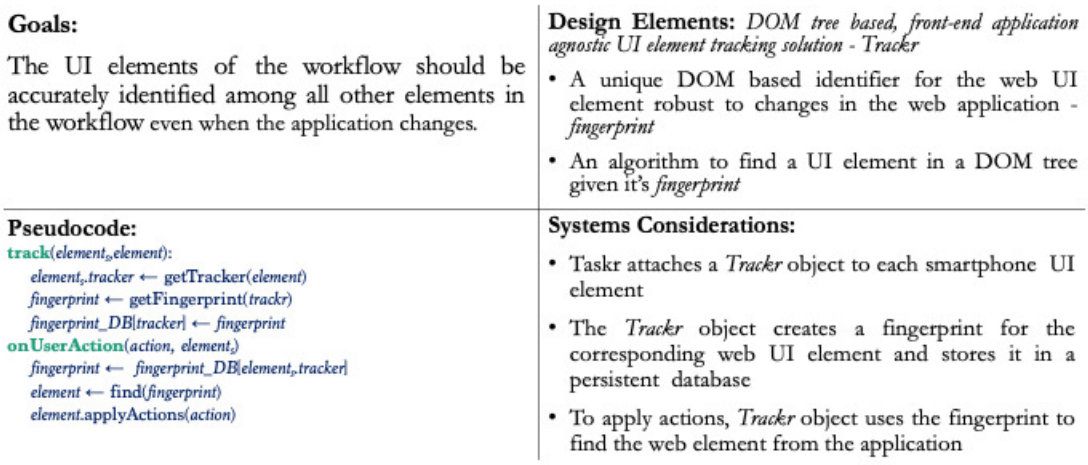


Figure 4.6: Overview of *Trackr*'s usage in *Taskr*

Table 4.1: Percentage of action elements with associated labels

Application	% of action elements with label
Sakai	65.4
Sharepoint	37.8
Utility Company *	77.3
Amazon AWS	95
Quickbooks	80.8

* *Name anonymized

Table 4.2: Different UI frameworks used by enterprise applications

Application	UI Frameworks
Salesforce	Ext Js [127]
Sharepoint	Bootstrap [129], XUI [130]
Quickbooks	Dojo [131], Express [132], New Relic [133]
Excelerate	Backbone [134], Require [135], Bootstrap, Underscore [136]
AtlasMD	D3 [137], Modernizr [138], Moment [139], NVD3 [140]
Utility Company	FancyBox [141], Bootstrap

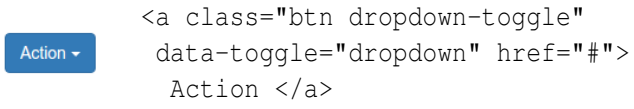


Figure 4.7: UI element from an external UI framework

UI elements - such as navigation bars, tabs, paginations, etc., on their websites. If this information is not correctly extracted, it will be difficult to comprehend the function that the UI element serves. For eg., without the label 'Username' next to a text field, the user cannot associate it with a place to enter her username.

Prompting the user to specify the nature of each UI element along with any associated labels at the configuration step can become very tedious for the user. On the other hand, an element's tag and the associated context can be inferred from its attributes and content inside the HTML tag. For example, if a UI element has the source <button id='submit_btn'> SUBMIT < /button>, its nature can be extracted as a button and the associated context as: label = 'SUBMIT'. However, for many elements, this extraction is not always possible. For eg., the label for input field can be declared via text surrounding that tag - Username: <input type='text' id='username'> To observe how often this is the case, we extracted all possible actionable UI elements from the landing pages of 5 popular enterprise applications and observed that on an average 28.74% of elements do not have any text present inside their HTML tags (See Table 4.1).

Furthermore, the Web is dynamic with new UI frameworks being designed each day to make web content more appealing to the end user. This problem is further aggravated by the presence of complex third-party UI frameworks[8]. For example, in Bootstrap, a button dropdown menu that has a HTML source shown in Figure 4.7 would be incorrectly classified as a link (from the 'a' tag). Therefore, *Taskr* uses a hybrid approach that not only obtains data from the source but also from the other surrounding tags, and by taking the user's help where such extraction is not possible. Using tag and attribute definitions from

[8]All the enterprise applications considered in this paper used atleast one third-party UI framework shown in Table 4.2.

Goals:

Cannot expect the user to code or script the specifications.

Design Elements: *Extract specifications from HTML source code for an element*

- A UI element can be described by the type of user interaction it allows – *nature* and the properties of the UI element – *context*
- Manually configured rules to map HTML source code of the element to *(nature,context)* for the element

Pseudocode:

```
gatherSpecifications (workflow):
    FOR element IN workflow:
        nature ← getNature(element.html.tag)
        context ← getContext(element.html.attributes)
    IF NOT confirmWithUser(nature,context):
            nature ← getNatureFromUser( )
            context ← getContextFromUser( )
getNature (tag):
// Rules to get nature from the element's tag code
getContext(attributes):
// Rules to get context from the element's attributes
```

Systems Considerations:

- Rule-based extraction for HTML5 standard
- At the configuration phase, Taskr prompts the user to confirm the automatically extracted specifications
- If the user disagrees with the automatically extracted specifications, Taskr requests the user to enter the specifications

Figure 4.8: Overview of *Taskr*'s data extraction

the HTML5 standard and from the complex UI frameworks, a list of rules for extraction is first created manually. For e.g., to get a label for an <input> element, the text within that element's tags is processed. When no text is found, the page source is be parsed to see if a 'label' tag for that input is present. At the configuration step, the extracted nature and context are displayed to the user. Whenever extraction using rules fails, the user is prompted to specify the nature and the context. Note that this is tractable as it only needs to be done once for every new UI element encountered. This is in sync with our goal of enabling the user to mobilize workflows, as the user best knows what to mobilize. Figure 4.8 shows an overview of data extraction with *Taskr*.

Translation to a mobile view:

Every UI element in the workflow selected by the user needs to be translated into the desired usage modality on the smartphone - native UI element for the smartphone app client and text for email, twitter, SMS, slack, etc. Enterprise web applications have been typically made for the desktop user. Therefore the UI elements are designed keeping in mind the input modalities of a desktop. Any translation of these elements should allow the same user actions (as in the desktop) to be performed on the modalities. Unlike traditional remote

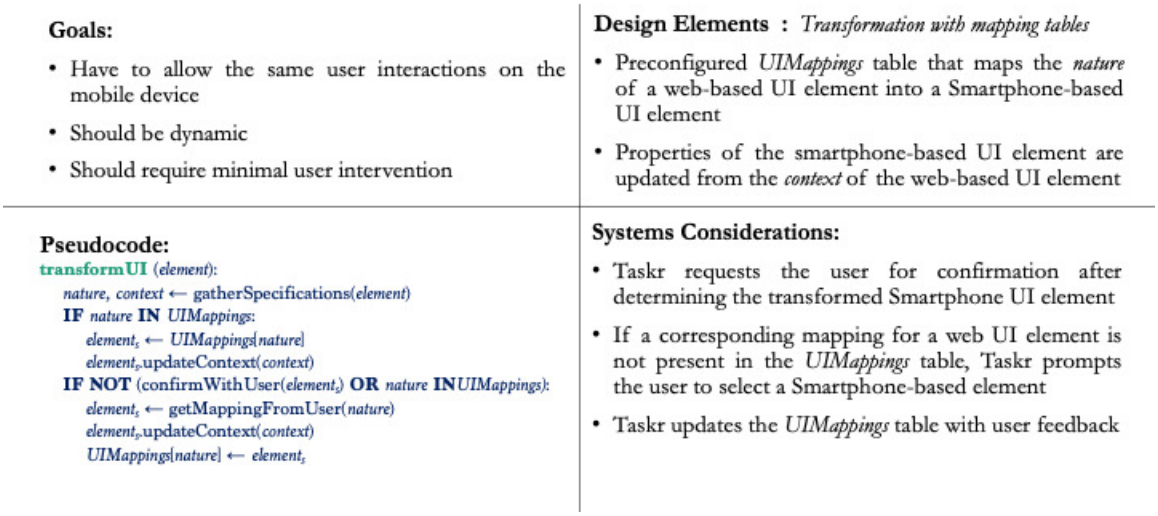


Figure 4.9: Overview of UI element translation in *Taskr*

computing that presents a remote view of the element as-is, and hence is subject to delays and unresponsive behavior, *Taskr* refactors the application view to suit the usage modality. *Taskr* uses a translation table that maps each UI element (including the ones from the third-party UI frameworks) to a corresponding native UI element (for the app) and also a text version for the other modalities. Each desktop UI element (including the ones from the third-party UI frameworks) is mapped beforehand to a smartphone native versions. The result of the translation is presented to the user during configuration. When the translation table does not contain a mapping for the selected UI element or if the result of the translation is not satisfactory, the user can manually specify the translation by selecting a type (e.g., text box, radio button, etc.) and a corresponding label. For different third-party frameworks, this phase can be combined with the information extraction phase and for every new element encountered this step needs to be done only once[9]. Figure 4.9 shows the translation in *Taskr*.

Note that *Taskr* translates each UI element of the workflow to one smartphone UI element. However, different platforms provide convenient macros that bundle user interactions across several elements into one interaction. *Taskr* divides the macros into the component

[9]Note that, the current version of the translation table covers most of the input elements from HTML5 standard.

UI elements and translates them individually to smartphone UI elements. *Taskr* can also be extended to consider these macros by asking the user to define any macros the user performs at the configuration stage. For the smartphone app, *Taskr* can create a new macro by bundling fixed parameter user actions at the last stage of the spot task into one UI element.

Mobile delivery and presentation

For every workflow stage, the translated versions of these elements have to be displayed on the mobile screen in a manner that enables the user to finish the task with minimal effort and should allow the user to easily comprehend the different actions to be performed to complete the task. Also, for ease of use, the mobile screen should preserve the sequence of actions performed while executing the workflow. For example, on a login page, the user typically enters the login information first, then the password and then clicks submit. At this stage, the elements should be organized in the order - username, password and submit. Displaying the password field before the username field can be non-intuitive for the user.

The task effort is directly proportional to the number of user actions needed to finish that stage. Specifically, task effort (τ) of a stage (S) is defined as - $\tau = \sum_{e \in S} A_{access}(e) + A_{perform}(e)$, where in, $A_{access}(e)$ is the number of actions to reach the element e (eg. scrolling till the element appears on the screen) and $A_{perform}(e)$ is the number of actions it takes to perform the function of e (eg: clicking a button). Each action on a smartphone can be - tap, longpress, drag, scroll, zoom, shrink, etc.

In the view of our goals of requiring minimal development effort and no assumption on the skillsets of the end-users, manually designing the layout for the workflow is not possible. Taking into account the simplicity of spot tasks and the inherent limits on the number of characters allowed, *Taskr* follows a fixed display template for every spot task. For the mobile app modality, *Taskr* divides the screen into three panes, and populates the READ elements in the first pane, the translated versions of the ACT elements in the second pane and two buttons 'SUBMIT' and 'CANCEL' in the final pane. The elements are displayed

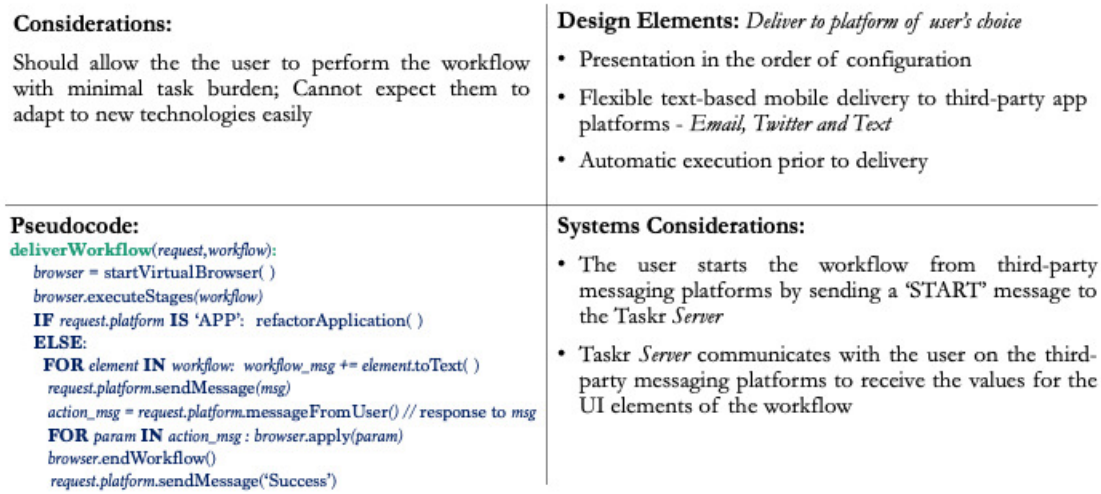


Figure 4.10: Overview of delivery and presentation in *Taskr*

in a list within the respective panes and in the order of their selection during the configuration phase to preserve the logical sequence of actions in the workflow. For the other usage modalities, a text blurb is constructed with the text version of the READ elements in the final stage followed by the labels of the ACT elements (one in each line) and sent to the user. To execute the workflow, the user can reply to this blurb with values for the ACT elements (one in each line and in the same order). Consider an example workflow of viewing payroll in Peoplesoft. One of the workflow stages is to solve a math problem to prove the user is not a robot (see figure 4.11). This stage involves 1 Read elements - math question, 1 Act element - the answer, and 1 Navigation element - submit. These elements are displayed in order. Figure 4.10 shows an overview of mobile delivery and presentation in *Taskr*.

User interaction and spot task execution:

If the user desires to execute any spot task, she has to first select an input modality to interact with. To complete the workflow using the *Taskr* mobile app modality, the user simply opens the app and selects the workflow among other workflows from the landing screen. Upon selection, the *Taskr* app informs the *Taskr-server* to start the workflow. The

Taskr-server executes all the stages of the workflow (except the last stage) using the fixed parameters recorded during configuration phase. The mobile app then presents a screen with READ and ACT elements of the final stage. When the user chooses input values for the ACT elements and clicks 'SUBMIT', the actions are sent to the *Taskr-server* where they are virtually executed on the application client.

On the other hand, to execute the workflow using other text based input modalities, the user has to send a start command ('#startworkflow') along with the name of the workflow to a fixed address (for email)/twitter handle (for twitter)/phone number (for SMS), etc. To get a list of configured workflows, the user can send a list command ('#listworkflows') to the server address to which the server replies with a list of workflow names. Once the server receives the start command, it begins executing the workflow. At the last stage, any READ elements and the labels for required ACT elements are sent to the user. The user can reply to this with the corresponding ACT values, which are then interpreted and executed on the application client. The user can also send an abort command ('#cancelworkflow') to cancel any currently running workflows.

Optimizations and Error Checking

Optimizations: All of the previous challenges deal with the user performing the workflow using a sequence of actions as on the desktop. Therefore, mobilization performance (time taken to finish the task) is limited by the application performance on the desktop. Given that the users are generally more tolerant to delays on the desktop than on the smartphone, any long delays can lead to a drastic decrease in the perceived user experience. Also, since the workflow is configured while the user performs it on the desktop, the number of actions taken to finish the workflow are the same as that of desktop. Therefore, the mobilized workflow needs to be optimized further to make the task execution even better than that of the desktop. This leads us to the following question - *In addition to mobilizing the workflow, what optimizations can be applied so that the overall user experience is better*

than that of the desktop?

The following are some examples of optimizations: (a) Handling common input values: Instead of the user typing/entering each input value, some common input values can be saved and presented as hints to the user when the workflow is performed for the next time. If these input values do not change, the process of entering them can be automated; (b) Auto Login: Several enterprise applications need the user to log-in before any workflow can be performed. Instead of performing login steps each time for a workflow, the login information in the form of a session cookie could be saved, allowing the user to auto-login the next time the workflow is executed; (c) Reducing the number of stages in a workflow: The number of stages in a workflow and the number of user actions needed to finish each stage currently are the same as that on the desktop. Whenever a stage does not require user input it can be automated; For example, a stage of the workflow that just has one navigate element can be performed automatically by the server without the user explicitly navigating. (d) Prefetching content to be used in the future: The delay in the mobile application is currently lower bounded by the delay on the desktop application. The delay can be reduced by opportunistically prefetching and rendering the next layout while the user is using the current layout; For example, while the user is typing, the result page can be fetched if it doesn't depend on the current input. (e) Performing spot tasks across devices: To allow for pausing and resuming of spot tasks from the smartphone, *Taskr* preserves the application session on the virtual browser as long as possible. *Taskr* also allows the user to switch devices in the middle of a spot task by recording any partially completed input fields on the first device and restoring them on the second device.

Error checking: Workflow execution can lead to errors due to several reasons: when page/element is no longer available, the user forgot to configure prerequisites (eg. login) for the workflow, the result of a stage of a workflow leads to another that has not been configured by the user, etc. When the application developer changes the application (say, removes a page), some elements of the workflow may no longer be found, thereby break-

Table 4.3: List of Worflows configured on enterprise applications

Application	Workflows	Application	Workflows
Amazon AWS	1. Create a security group 2. View service status 3.View instance status 4. View account balance 5. Create new volume	Peoplesoft	1. View the latest salary amount 2. Add direct deposit account 3. View year to date earnings 4. Get balance vacation hours 5. Update contact information
Sharepoint	1. Get the next task deadline 2. Create a task and assign it 3. Edit a wiki page 4. Sync the website 5. Share a project	Salesforce	1. Get Quarterly net performance 2. Create a poll for followers 3. Get information on the top deal 4. Create a new campaign 5. Create an open lead
Quickbooks	1. Add a new customer 2. Add a new service 3. Get net profits/loss this month 4. Add a new vendor 5. View income report	Sakai	1. Edit a Wiki page 2. Change page permissions 3. Add a participant to a Wiki 4. Create a new group 5. Check a Wiki's last edit owner
Utility	1. View latest balance 2. View usage history 3. Pay latest balance 4. Get usage in current month 5. Get plan expiration date	AtlasMD	1. Add a patient 2. Add a note to the pharmacist 3. Add a new pharmacy 4. Close office next monday 5. Add a pricing tier
Excelerate	1. Retrieve VIN of the vehicle 2. Add a service request 3. View vehicle registration information 4. Request an insurance card 5. Report mileage		

ing the workflow; The presence of these errors leads to the following challenge - *How to handle any errors that may be encountered while the user is performing a workflow, which are a result due to drastic changes in the application flow or due to user error?* The mobilized application gracefully shuts down upon encountering any error. A display message which corresponds to the cause of the error is prompted the user. This allows the user to re-configure the workflow correctly.

4.4 Evaluation

Prototype

We implement a proof of concept prototype of *Taskr* with which users can easily mobilize spot tasks and execute them through three different usage modalities - app, Twitter and Email (see Figures 4.11a, 4.11b, 4.11c, and 4.11d). Within this prototype, the *Taskr-*

recorder is a Javascript browser extension for Google Chrome. In addition to observing the user action while the user is performing the spot task, the tool also allows user to manually select elements, extracts fingerprints and transforms using rules.

The *Taskr-server* is written in python and deployed in the Amazon EC2 cloud. When the user selects a spot task, it instantiates a headless Chrome browser and attaches a Selenium automation driver to it. Upon receiving any user actions performed on the *Taskr-client*, it executes them on the browser through selenium. For the Twitter usage modality, the server uses Twitter Direct Messaging APIs to filter out appropriate commands from its Twitter stream and to send responses to the user. For the Email usage modality, the server monitors its email mailbox for any emails with commands using Python's imaplib. Any response to be sent to the user is handled by smtplib.

Finally, the *Taskr-client* is implemented as an app for Android OS. The landing screen of the app lists all spot tasks that have been mobilized, organized by the application name. Upon selection of a task, *Taskr-client* renders the UI elements of final stage of the workflow using translations constructed during configuration phase. Any user actions on these native elements are reported back to the *Taskr-server*. The user can either execute the workflow (i) using *Taskr-client*, (ii) by sending a direct message with an appropriate start command to the server's twitter handle, or (iii) sending an email with the subject containing the start command to the server's email address.

User Study:

We mobilize spot tasks in 9 enterprise applications using *Taskr* in the following categories - Learning Management System (Sakai [128]), Human Resources Management (Oracle Peoplesoft), Collaboration (Sharepoint), Customer Relationship Management (Salesforce CRM), Accounting (Quickbooks [27]), Cloud Management (Amazon Web Services), Billing portal (A utility company website - name anonymized), Electronic Health Record (AtlasMD) and Fleet Management (Element Fleet). We configure five workflows from each

of these applications representing typical daily usage patterns of employees. For brevity, we only show workflows from four of these applications in Table 4.3. We configure each workflow through the taskrt. During the configuration phase, we select the elements on the page, mark their category as read/write/navigate. If a label is not picked by default, we type an appropriate label when prompted. Once the workflow is configured, it is automatically uploaded to the server hosted on Amazon EC2 cloud.

We then start the *Taskr-client* on a Google Pixel smartphone (Android 7 Nougat) and the *Taskr-server* on an Ubuntu Server hosted on Amazon EC2 cloud instance. The instance is of type m4-2xlarge with 4 vCPUs and 32 GB of RAM. We subsequently execute each of the workflows on the *Taskr-client*, Chrome browser on the smartphone, and a Chrome browser on a desktop. Whenever the workflow cannot be performed using the mobile web version of the application, we load the desktop page of the application on the mobile browser to complete the workflow. Figures 4.12 shows the number of actions taken by user to perform the workflows in Table 4.3 for the *Taskr-client*, the Chrome Browser on the smartphone, and the Chrome Browser on the Desktop. We observed that, on an average, the workflows on *Taskr-client* take 40.67% fewer actions compared to the desktop browser and 38.19% fewer actions compared to the mobile browser.

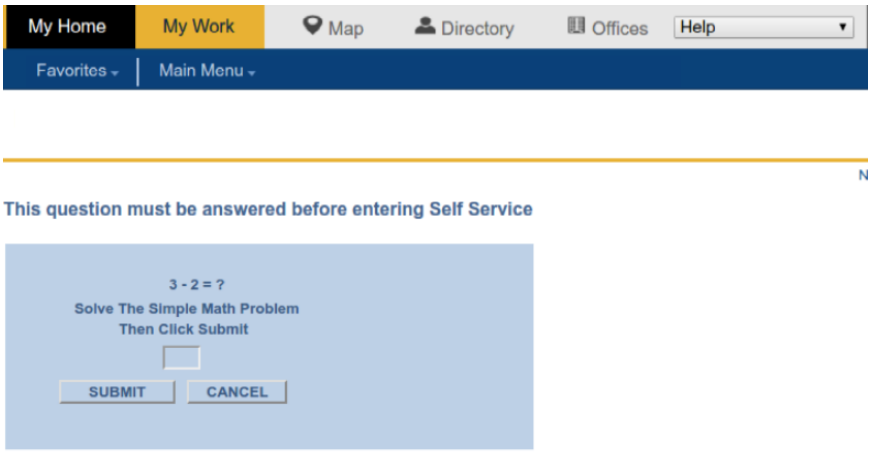
We also evaluate *Taskr* using subjective experiments on 15 volunteers [10]. We selected the following 5 workflows from 3 applications - Sakai (editing a wiki page, changing permissions of a site and adding a participant to a site), Amazon AWS (#4), and Peoplesoft (#2). Each volunteer performed the workflows on three platforms (*Taskr* client, Desktop and Mobile browser) in a random order. The order of the platforms on which these users performed the workflows was also randomized. The volunteers were then asked to answer the following 7 questions rating each of the platforms.

- On a scale of 1 to 5, how satisfied are you with the application?
- I am satisfied with the number of steps it took to finish the workflows

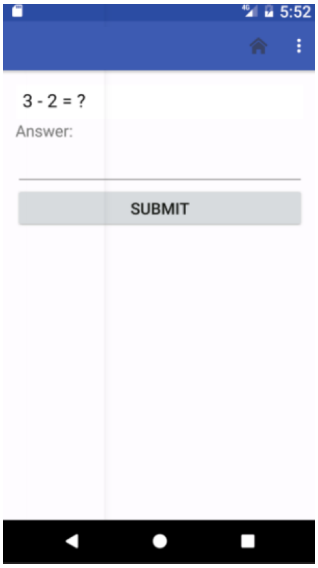
[10]The volunteers were mostly university students within 22-30 year age group

- Information presented on the screens was easy to comprehend
- How easy is it use and figure out the app?
- Navigation: Is moving between screens logical/accurate/appropriate?
- How long does it take to perform the workflow on this application?
- Would you recommend this application to people who might benefit from it?

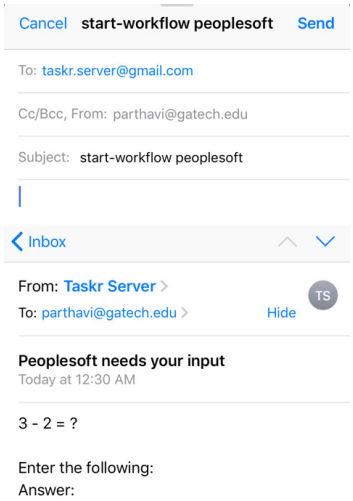
The users were then asked to answer these questions by choosing one among 5 options. The options presented for the users were based on a likert-scale with likert-type responses. For example, the options for Question 1 would be - strongly disagree, disagree, neither agree nor disagree, agree, and strongly agree. Each option has a score corresponding to it (from 1:worst to 5:best). Figure 4.13 shows the % of total responses across the scores from the users in a stacked graph for three of the questions - *How satisfied are you with the application?* The responses to other questions follow similar trends. The users consistently rated *Taskr-client* better than the other two platforms for all the questions. For example, 100% of the users were satisfied (score $>$ 3) for *Taskr*. On the other hand, only 66.67% of users were satisfied with the desktop experience and 13.33% with the mobile experience. The desktop was rated the better in general than the mobile, due to the user's familiarity with the application on the desktop.



(a) Peoplesoft on Desktop



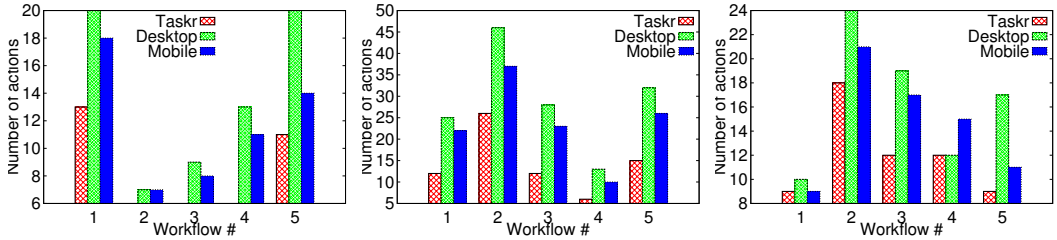
(b) *Taskr-client*



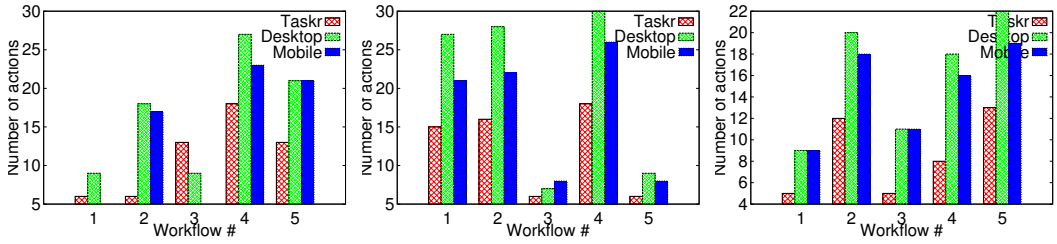
(c) Email

(d) Twitter

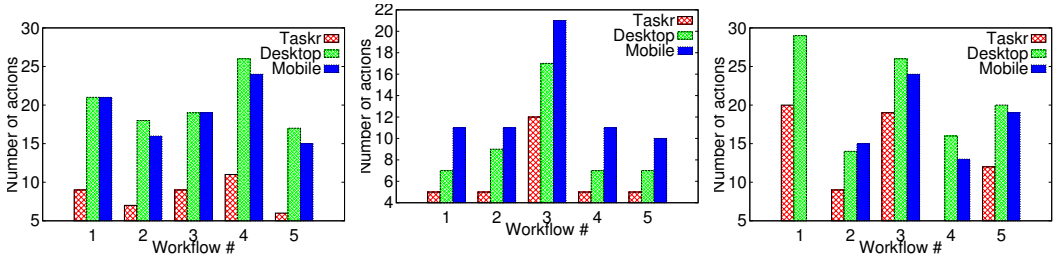
Figure 4.11: *Taskr* prototype for a test workflow on Oracle Peoplesoft



(a) Amazon Web Services (b) Peoplesoft (c) Sharepoint

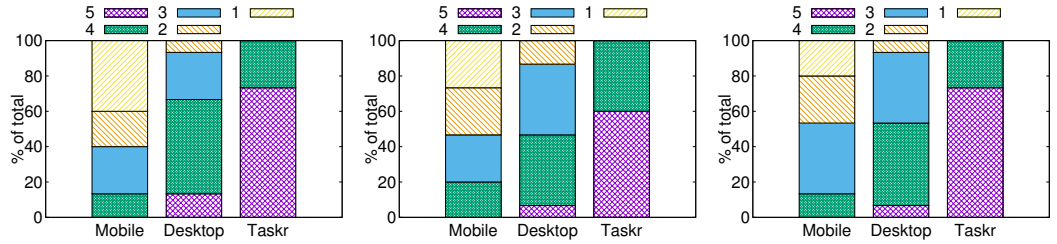


(d) Salesforce (e) Quickbooks (f) Excelerate Fleet



(g) Sakai (h) Utility Company Website (i) AtlasMD

Figure 4.12: Number of actions taken to perform workflows on enterprise applications



(a) Question 1 (b) Question 3 (c) Question 5

Figure 4.13: Mean Opinion Score from volunteers

CHAPTER 5

TRACKR: RELIABLE TRACKING OF UI ELEMENTS WITHIN WEB APPLICATIONS TO ENABLE ROBUST APIFICATION

5.1 Introduction

A relatively recent trend in the domain of web applications is to APIfy applications so that evolutionary secondary services may be built upon them seamlessly. In fact, the increasing adoption of web service frameworks such as REST and SOAP in building web applications is in line with allowing for such seamless extensibility. A simple example of APIfication of web applications is Google Maps. While Google Maps is itself a popular application used by users to obtain navigation information, other applications can also leverage the API exposed by Google Maps such as those for directions, distance, elevation, geolocation, roads, and time zones. The APIs can be used by any application over HTTP, allowing for faster integration of mapping and navigation intelligence into those applications. Well known applications such as AirBnb, Expedia, Allstate Goodhome, NYTimes, 7-Eleven, and Runstatic all rely on Google Maps APIs [142]. Most popular web-based applications such as Gmail, Salesforce, Twitter, etc., have their own APIs that other applications can leverage.

While applications can indeed be built ground up to support APIs, an interesting problem is how web applications not built in such manner can still be retroactively APIfied. Such a scenario occurs under two different conditions: (i) the applications are legacy applications that pre-date the APIfy movement, but still command considerable usage wherein APIfication will have tremendous value; and (ii) the applications are built by a vendor who does not have any explicit business or technology motivation to expose APIs to third party developers (even if they do exist on the backend). The second issue is more pertinent as ex-

posing APIs for a web application does come with its own costs such as ensuring security, incurring maintenance overheads, facilitating monitoring and monetizing, and provisioning for scalability. A more nuanced version of the aforementioned problem is when a third-party developer needs a certain functionality offered by the web application but not exposed through an API. One approach to APIfy is to rewrite the underlying software for the web application to expose APIs. However, such a strategy incurs the burden of both the redevelopment of the software, and the redeployment of the application. Hence, the rebuilding-based strategy is an expensive process and is quite undesirable.

A different strategy to APIfying a web application relies on front-end only techniques to create APIs. Using a combination of automated navigation, intelligent acting, and content scraping, front-end APIfying techniques create APIs without requiring any changes whatsoever to the application backend. Consider the simple example of a thermostat web application (that could control a smart thermostat inside a home) that requires the current temperature for a zip code. Regardless of the APIs supported by a service such as weather.com, a front-end APIfying approach can create APIs for weather.com that will provide the current temperature for a zip code purely by navigating to weather.com, entering the zip code in the search bar, and scraping the temperature information from the resultant view. The salient advantage of this strategy is the non-dependence on backend changes. This is certainly less expensive. More importantly, APIfying an application is no longer dependent on the vendor who created the application. Third party developers can as easily create APIs for it.

It is such front-end based APIfy strategies that we consider in this paper. Specifically, such strategies rely on a fundamental building block - the ability to uniquely identify and track front-end UI elements on the web application. For example, in the smart thermostat use-case, consider that the temperature UI element is uniquely identified on the resultant view on weather.com. The thermostat application will now rely on an API that reads the temperature from that specific UI element on weather.com. What happens if the

weather.com changes in a manner that impacts the temperature UI element? There are indeed changes that should break the API a good example would be if weather.com removes the temperature UI element. However, there are a variety of changes including the temperature UI element moving to a different location, new UI elements introduced on the page, other (non-relevant) UI elements removed from the page, attributes of UI elements such as color, size and labels change, etc., that should not break the API. This challenge is the focus of this paper.

What makes the challenge non-trivial is that UI elements within web-applications, organized in a DOM tree, *do not have distinct permanent identifiers that remain invariant across application changes*. Thus, only relative identifiers (e.g. path from DOM tree root) can be relied upon to uniquely identify UI elements. These relative identifiers are vulnerable to even minor changes to the application that impacts the DOM tree in some manner. In this context, we present *Trackr*, a UI element tracking algorithm that improves the robustness of APIs created atop web applications multi-fold. At a high level, *trackr* uses the concept of *quorum fingerprinting* that determines the identity of a target UI element based on its relative paths from other nodes in the DOM tree that have an attribute ID. We then argue why such an approach by itself remains insufficient to handle the different types of possible changes to the web application. We then present multiple optimizations to the baseline quorum fingerprinting including resilient path construction, progressive patching of fingerprints, and localized fingerprints as fail safes. We show using popular web applications such as Salesforce, a PeopleSoft application, a SharePoint application, and a Sakai application that *Trackr* can improve the identification of a target UI element multi-fold compared to standard mechanisms. We then present three different use-cases that rely on APIfied web applications and discuss how they benefit from *Trackr*.

The rest of the chapter is organized as follows: Section 2 presents background and motivation for *Trackr*. Section 3 outlines the *Trackr* design. Section 4 presents evaluation results and Section 5 discusses use-cases where *Trackr* can be used to deliver better perfor-

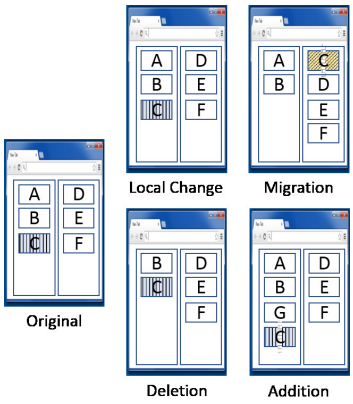


Figure 5.1: Possible changes in a web application

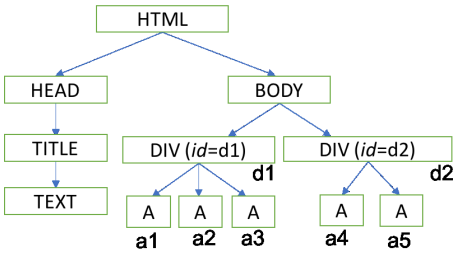
mance. Finally, Section 6 discusses a few issues with *Trackr* and presents key conclusions.

5.2 Background and Motivation

5.2.1 Web Applications and DOM Trees: A Primer

Web applications are gaining popularity today as they are platform independent, easy to deploy and have a well established development infrastructure. They provide a convenient way to deliver different functionalities to the user with minimal development costs. A web application can be typically accessed on any browser through it's URL. A web application is a collection of web pages, most of which are rendered on the browser as HTML documents. The underlying data structure for an HTML document is a tree called the Document Object Model (DOM). The DOM tree defines the layout of the application. Each tag from the document is an element of this tree. The tree is rooted at the <HTML> tag. Any nested tags within a particular tag are children elements of that tag. Fig. 5.2 shows the DOM tree for a simple HTML document in Fig. 5.3.

Each tag in the HTML document can be classified as: (i) meta (e.g., meta, link, script, etc.), (ii) formatting (e.g., p, br, bf, etc.), (iii) layout (e.g., div, table, etc.), or (iv) action (e.g., a, button, textbox, etc.). After the tree is rendered, the effect of these tags can either be visible (layout and meta tags) or invisible (action and formatting tags). All




```
[baselinestretch=0.7]
<HTML><HEAD>
<TITLE>title</TITLE>
</HEAD><BODY>
<DIV ID=d1>
<A href=l1></A>
<A href=l2></A>
<A href=l3></A></DIV>
<DIV ID=d2>
<A href=l4></A>
<A href=l5></A>
</DIV></BODY></HTML>
```

Figure 5.2: DOM Tree **Figure 5.3: HTML Source**

modern browsers allow the DOM tree to be accessed through Javascript DOM API[143]. For example, the method $getElementById(id)$ returns the element with an attribute ID=id, $getElementsByTagName(tag)$ returns an array with elements whose tag name = tag, etc. Also, as users interact with elements their appearance defined by their HTML attributes can change even though the layout remains the same. For example, when a user clicks on a checkbox, the attribute 'checked' is toggled.

UI Element Identifiers

A tag can have some HTML attributes associated with it. For example, the tag <A href="link1"> has one attribute href. The attribute values need not be unique for the tags. One exception to this rule is the attribute ID. Therefore, the value of an HTML attribute ID is a *globally unique identifier* for that element. While such an identifier is highly desirable for an element, it is not always available. For example, in the Salesforce web application, only 19% of all elements have an ID declared. Also, even though the value of the ID is unique in an instance of a DOM tree, it is not necessary for it to remain constant across multiple instances of the application. ExtJS[127] is a popular JS library that creates different ID values at different instances. On the other hand, using the attributes contained within the tags of an element, an *attribute based identifier* can be constructed. However, this identifier is not unique as it is not necessary for an element's attributes to be unique in a DOM tree.

For example, in Salesforce, only 16% of elements have a unique set of attributes. Given that the element's own attributes will not help in its identification, the next logical direction is to consider identifiers that are relative to some property. When this property is *relative to the element's local context* within the DOM, the identifier again is not unique. For example, a local identifier consisting of an element's parent, immediate siblings and children is only unique for 13% of the elements on Salesforce. On the other hand, identifiers that describe an element *relative to a unique global property* within the DOM are unique. Some examples of such IDs are - Path from the root, Path from all elements with IDs, Coordinates from the top left corner of a page, Path from the body element, etc. Such an identifier can be constructed for every element within the DOM. Also, given an identifier and any DOM tree, at most only one element can be found with the same identifier.

On the nature of changes

While an element's global relative identifiers can uniquely identify it given a DOM tree, it is not necessary that they remain constant even when the DOM tree changes. For example, when the dashboard of Salesforce application is reconfigured to add a new 'messages' section, all the elements that immediately follow this section (e.g., recently viewed) will have their global relative identifiers changed i.e. the paths to these elements in the DOM tree get altered. Web applications are dynamic and rich today, wherein the DOM trees not only change due to the developer modifying the application, but also because of user interactions[144]. Any user action can either modify the DOM tree or can lead to a new webpage with a new DOM tree. For example, clicking on a list will modify the DOM tree by introducing option elements into the tree, where as clicking on a link will load a new webpage. A web application can undergo several types of changes such as layout modifications, content updates, appearance changes (either by the style attributes of elements or when a UI library is updated) and code changes. These changes affect the underlying DOM structure in one of the following ways (see Fig. 5.1): (i) *Local changes:* These are the changes

wherein only attributes within an element are changed leaving the DOM tree intact. For example, the change in color of a link after a user clicks on it; (ii) *Insertions:* These are changes wherein a new element is inserted into the DOM tree. For example, when a user creates a new task and it is added to the list of all tasks; (iii) *Deletions:* These are the changes wherein an element is deleted from the DOM tree. Any children of this element are inserted at the element's position before deletion. For example, when container DIV is deleted and all its children are moved back into the parent DIV; (iv) *Migrations:* These occur when an element (and any descendants) moves to any other position in the tree. For example, when a user decides to reorganize a dashboard, say by moving the list of tasks to another location within the dashboard; Consider an example wherein a simplistic DOM tree shown in Fig. 5.2 changes to the tree shown in Fig. 5.5. The change in attribute ID value of element d_1 to d_3 is a local change, the addition of p_1 is an insertion, absence of a_3 and a_4 are two deletions, and movement of a_2 is a migration. All other changes can be expressed as a combination of these categories.

5.2.2 Problem Definition, Scope, and Goals

In this paper, we target the problem of developing an algorithm to reliably track UI elements of a web application across several instances of the application. Note that web-applications innately do not need to have distinct permanent identifiers for the UI elements. Hence, UI elements can be identified only by a relative identifier constructed based on some property of the underlying DOM tree. Hence, the problem involves creating a unique identity (called the *fingerprint*) for the UI elements that remains robust even as the application changes. We only consider web applications that are rendered as HTML documents on the client browser due to their dominance in the web application ecosystem[124]. In this chapter, we treat web elements as containers of content, and not as content itself. For example, in a list of recently viewed headlines, when a particular headline content originally at the top of the list moves to a different position, the web element corresponding to the top position in the list hasn't

moved but the content it carries changed. On the other hand, if the list of headlines as a whole is moved to a different location on the page, we assume that the web elements have moved. Furthermore, the framework should be able to track any element belonging to the DOM tree as a whole. Tracking parts of a node individually is beyond the scope of this paper. For example, if there is a paragraph of text declared as a node of the DOM tree, the framework should be able to track paragraph as a whole, and not individual words within it. Also, given a fingerprint, tracking should only return an element if it is present in the application. If the element is deleted from the DOM Tree either by the developer modifying the application or the user configuring the application, tracking should return an empty pointer.

The problem considered in this chapter can be formally stated as - *Given a web application A with a DOM tree* τ*, how can a unique fingerprint for any given web element* $e \in \tau$ *be created, such that the fingerprint can effectively be used to identify the element e in a different instance of the DOM tree* τ'.

Any algorithm for tracking UI elements should satisfy the following goals: (i) The algorithm should be robust and withstand a wide range of changes within the DOM structure; (ii) It should be able to track elements with only the information available from a typical web-application and make no assumptions about any additional resources from the web-applications, especially from the application developers; and (iii) Finally, it should be application agnostic.

5.2.3 Problem Relevance and Significance

Given the rising popularity of web applications, there are several secondary web services available that extend the functionalities provided by the (primary) applications. These applications usually rely on the APIs provided by the web applications to access their features. A common goal among these secondary services is to observe some variables from the web application(s) and act on the them to provide the necessary functions. To explain the rel-

evance of the problem considered in the paper, we discuss three such secondary service use-cases that rely on accurate and reliable tracking of UI elements within a web application. For each of these use cases, the lack of a reliable tracking algorithm leads to a significant increase in task burden.

(i) Automation:

Automation services like Selenium[145] help users programmatically perform a sequence of tasks within a web application, so to eliminate the task burden of performing them manually. To automate a particular action on a web element using Selenium, a user has to write a script that declares how to access the web element using simple Javascript DOM access methods and specifies the type of action to be performed. However, as the application and the corresponding DOM tree change, it is possible that the access methods mentioned in the automation scripts fail to access the correct element. In this case, the user has to manually rewrite the automation scripts to access the elements in the modified DOM. This can be burdensome. An accurate element tracking algorithm can effectively eliminate this burden.

(ii) Macro-Creation:

With services like IFTTT (If this then that) [146] users can create macros to observe certain variables within a web application, create triggers when the variables satisfy some conditions and perform specific actions on a different web service. For example, a user can create a macro that tracks a package and emails a public transit schedule to reach home in time to collect the package. IFTTT relies on APIs provided by the web applications to create triggers and perform actions on their data. However, given that a vast majority of applications do not expose a compatible API, the users are restricted to using only a limited number of web applications. With the availability of an accurate tracking algorithm, third-party services like IFTTT can reliably access application data from their DOM structure, independent of any support from the application itself.

(iii) Application Mobilization:

Application mobilization services allow the employees to use their smartphones to complete the tasks that were originally performed on a desktop. Of all mobilization strategies, application refactoring has minimum development and deployment costs. This involves hosting the application as-is on a cloud and presenting the users with an optimized native UI on their smartphones. Any actions on the native UI are then executed on the original UI in the cloud. Capriza [147] is an example of a refactoring based mobilization service. A critical step in refactoring is to map any actions from the smartphone native UI to actions on the original application UI. This requires reliable tracking of the UI elements in the original application even as the application changes. If the tracking is inaccurate, the mapped actions are possibly performed on the wrong element leading to failures.

Later in Section 5.4, we revisit these use-cases to demonstrate how *Trackr* framework can be integrated within them.

5.2.4 Related Approaches and Performance Analysis

When the services relying on the application's front-end (such as the three examples above) fail due to a change in the application, they have to be reconfigured again. This can lead to increased task burden and more costs.

Prior work:

The problem of reliably fingerprinting UI elements within a web application has been explored in the past in different contexts. XPath[86] is a widely adopted standard with syntax to describe elements within an XML/DOM tree. Using XPath syntax, a path for traversal within a DOM tree can be specified between two elements. For example, $//html//body[1]$ is the XPath expression to reach *body* by traversing to html's second child. However, XPath only provides a syntax and it is upto the developer to create a fingerprint with it. Several optimizations[87, 88] have been proposed to interpret XPath. In [89], an element's path

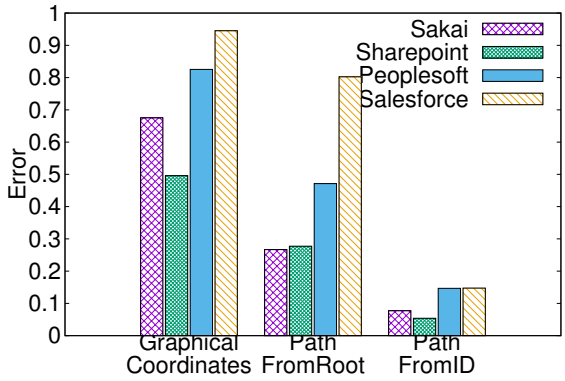


Figure 5.4: Performance of existing fingerprints

from the root of the DOM tree is used as one of its features, but in the context of enhancing mining. [90] uses the shortest path from the nearest ancestor in the DOM tree with an HTML attribute ID as a fingerprint. Here, the context is to record user actions. [91] uses path from the root in conjunction with parent and immediate siblings to identify an element for information extraction. In [92], the authors propose using subtree information for each element in a DOM path. These fingerprints assume a consistent DOM for the web application, which does not hold true in reality. We later show that these single-path based fingerprints do not perform well in dynamic scenarios. [93, 94], use visual features of the page to learn and extract templates for elements. [94] also uses visual features to generate a layout structure for a webpage analogous to DOM. This layout structure can then be leveraged to create fingerprints. However, generating fingerprints based on visual features is not feasible for a majority of secondary services as it not only requires a large amount of annotated training data but also takes a lot of time.

Performance of related approaches:

One obvious candidate for an element's fingerprint are it's coordinates w.r.t. top left corner of the webpage (*Graphical Coordinates*). However, it is not robust and small changes within the application can easily break it. For example, if the title of a news article is updated to a longer sentence, the *Graphical Coordinates* of all the content that surrounds

it will change. On the other hand assuming that the application has a consistent DOM, an element's position within this tree can serve as it's fingerprint. Since all elements cannot be directly accessed without a HTML attribute ID, the element's position can be obtained by traversing the shortest path from the root of the tree to the element (*Path From Root*). However, when the layout of the tree changes, it is possible that this path will lead to a different element. The probability of layout changes affecting the *Path From Root* can be lowered by considering path from an anchor element closer to the given element than the root. The only elements in the tree that can be directly accessible are elements that have an attribute ID. Therefore, the path from a nearest element with an ID (*Path From ID*) along with the ID value can act as an element's fingerprint. To study the robustness of the three fingerprint candidates - *Graphical Coordinates*, *Path From Root*, and *Path From ID*, we downloaded the home pages (after login) of four popular web applications- Sakai, Sharepoint, Peoplesoft and Salesforce and artificially introduced changes within their DOM structures. We first randomly select 30 elements from the DOM to track and introduce changes to the DOM (each element has a 0.5% chance of changing). We then find these elements in the modified DOM tree using the three fingerprint candidates. More details on this experimental setup are explained later in Section 5.4. Fig. 5.4 shows the ratio of elements whose fingerprint fails to find the elements within the modified tree. On an average, the error rates are 0.73, 0.44 and 0.11, for *Graphical Coordinates*, *Path From Root* and *Path From ID*, respectively. These experiments lead us to a few key insights: (i) DOM based fingerprints that leverage the application layout perform better than pixel based graphical coordinates; (ii) *Path From ID* has a much lower error rate compared to *Path From Root*. This can be attributed to the shorter length of *Path From ID* as shorter paths have a lower probability of being affected by changes; (iii) Even though *Path From ID* performs much better than other fingerprints, the error rate is still very high and unacceptable. This calls for the development of a new fingerprint algorithm.

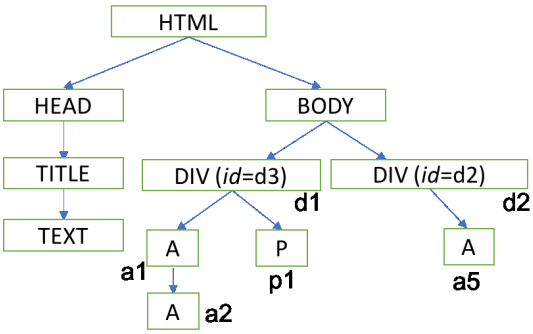


Figure 5.5: Changed DOM Tree

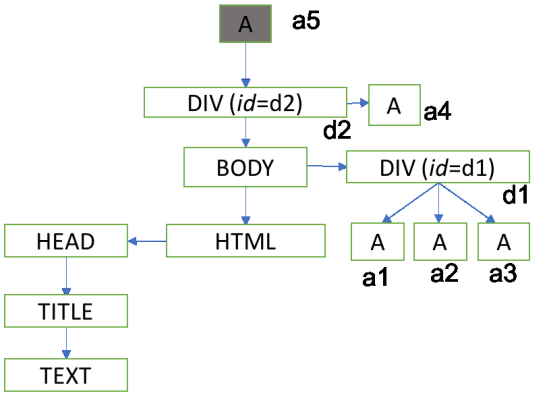


Figure 5.6: Quorum Tree of a_5

5.3 Trackr: Fingerprinting Algorithm

5.3.1 Architecture Overview

We design *Trackr* as a passive browser extension that secondary web services can rely on to track any number of UI elements from a web application. *Trackr* extension exposes two key functions - *Trackr.track(element, tname)* and *Trackr.find(tname)*. Any web service can use the *track()* function to track a certain element by passing the element's current handle (Javascript DOM object) - *element* and a name for the tracker - *tname*. *Trackr* then extracts a unique identity (*fingerprint*) for the element and adds it to a database stored in the browser's persistent storage. The fingerprints in the database are indexed by the URL of the web page from which the fingerprint was extracted and the name given to the tracker (*tname*). At every subsequent visit to the page, *Trackr* updates the fingerprint to reflect any changes within the DOM since the last time the fingerprint was computed. Using *Trackr*'s $find()$ function and the tracker name *tname*, the service can request a current handle to the tracked element. *Trackr* then retrieves the fingerprint from the database and uses it to find the element within the DOM tree.

5.3.2 Quorum Fingerprinting

In Section 5.2, we evaluated the performance of three simple fingerprints and observed that single-path fingerprints are insufficient to reliably track elements in dynamic web applications. Also, recall that the fingerprinting algorithm cannot assume any other information from the web applications except it's DOM structure and elements can only be identified relative to some other property of the DOM tree. Therefore, instead of just considering the position of the node in the DOM tree w.r.t. one other element (root or node with an ID), *Trackr* adds redundancy into the fingerprint by considering the position of the node with respect to all elements with an attribute ID. The key insight is that even if some portion of the DOM tree changes between two instances, a majority of the tree remains intact. Therefore, by considering position w.r.t. several anchors and using a simple majority rule to identify the element that matches most of these positions in a modified tree, *Trackr* creates a robust fingerprint. We call this principle *quorum fingerprinting*.

To construct a quorum fingerprint $Q.FP()$ for an element e in a DOM tree τ, *Trackr* reshapes the DOM tree so that it is now rooted at e (*quorum tree*). Reshaping is done by first inverting the shortest path from the element e to the HTML root, so that e is now at the root position of the new tree. *Trackr* then appends all the other elements as children to their respective parent nodes from the old tree. Fig. 5.6 shows the quorum tree for the node a_5 from the example shown in Fig. 5.2.

Using the quorum tree, *Trackr* computes the shortest path from all elements with an attribute ID to the root of the quorum tree. Therefore, $Q.FP(e) = (ID(a), SP(a,e)) \forall a \in \tau$ and a has an attribute ID, where $ID(a)$ is the value of attribute ID for a and $SP(a,e)$ is the shortest path between a and e in the quorum tree. Shortest path $SP(a,e)$ is computed by traversing the quorum tree upwards from the element with ID until its root is reached. For each element encountered in the traversal, the element's name along with the index w.r.t. to its siblings (in the original tree) is recorded, i.e. given an element e, and it's quorum tree $Q(\tau, e)$, $SP(a,e) = [(name(e'), index(e')) \forall e'$encountered in the traversal to e], where

$index(e')$ is the index of e' w.r.t it's siblings in the original tree τ. For example, the element a_5 has a quorum fingerprint $Q.FP(a_5) = (d_1, [(BODY, 2), (DIV, 2), (A, 2)]), (d_2, [(A, 2)])$.

In order to find an element in another instance of the DOM tree τ', *Trackr* compares $Q.FP(e)$ to the quorum fingerprints of all other elements of the same type (as e) in τ'. For each element e' of the same type in the modified tree, *Trackr* uses Algorithm 1 to compute a score that reflects how many of the paths in e's fingerprint match with those of e'. The element with the maximum non-zero score among all other elements is e's counterpart in τ'. For example, element a_1 can be identified in the modified tree (Fig. 5.5) using the quorum fingerprint computed from the tree in Fig. 5.2. Even though it's nearest anchor DIV d_1's ID has changed, the path from the other anchor element d_2 remains intact in the modified tree. Figure 5.7 shows an overview of the fingerprinting algorithm of *Trackr*.

Given that an element's quorum fingerprint contains the traversal paths from all other elements with an attribute ID to the element, it can be expensive to compute when the DOM tree has many anchor elements with attribute IDs. For these situations, *Trackr* provisions for a flexible parametric quorum fingerprint that limits the number of anchor elements considered. This limit can be tuned by the developer using *Trackr* to balance the tradeoff between the accuracy of tracking and the complexity of computing paths. If a limit K on the number of paths is set, the quorum fingerprint of an element *em* only contains the paths from the closest K elements within the DOM Tree to *em*.

Algorithm 1 Baseline Algorithm

```
1:  procedure match_fingerprint(Q.FP(e), Q.FP(e'))
2:      score ← 0
3:      for id ∈ Q.FP(e) do
4:          if id ∈ Q.FP(e') then
5:              P1 ← Path corresponding to id in Q.FP(e)
6:              P2 ← Path corresponding to id in Q.FP(e')
7:              if P1 == P2 then score ← score + 1
8:              end if
9:          end if
10:     end for
11:     return score
12: end procedure
```

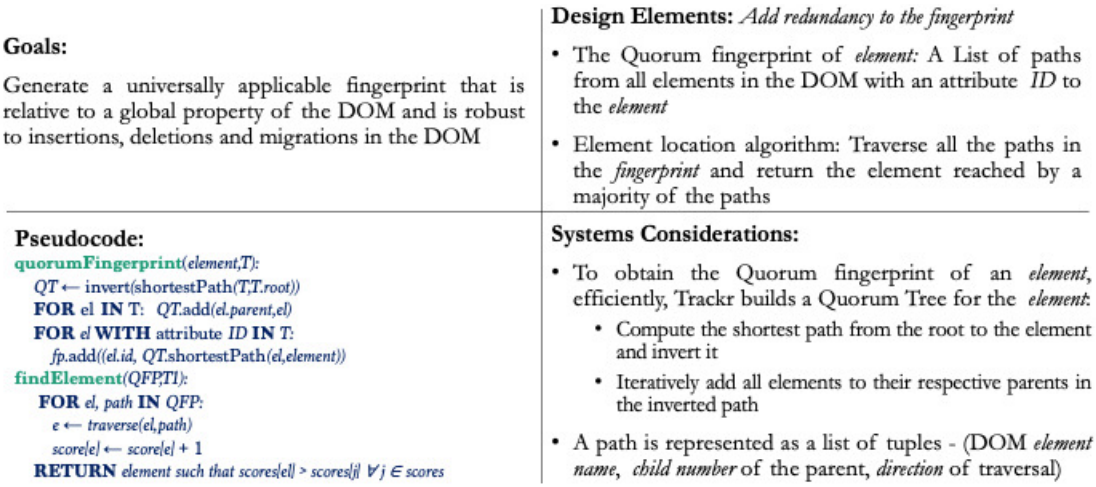


Figure 5.7: Overview of *Trackr* with Quorum Fingerprinting

5.3.3 Fingerprinting Optimizations

Through the principle of quorum fingerprinting, *Trackr* increases the immunity of the fingerprint to DOM changes. We now describe five different optimizations that are progressively applied to the baseline algorithm to make it more robust.

(i) $Trackr_{PR}$ **Path Resiliency:** Even though the baseline fingerprint described earlier is robust to secluded changes in the DOM tree away from the element, the presence of many changes in the vicinity of the element can still break the fingerprint. For example, element a_5's quorum fingerprint is insufficient to find it in the modified tree, as its index w.r.t to its siblings has changed. To counter this problem, *Trackr* adds resiliency to how paths are calculated. Instead of just using the name of an element and the index (w.r.t. its siblings) to differentiate it from its siblings, *Trackr* computes three parameters from the original tree τ - (i) l: the number of siblings to the left of the node, including the node; (ii) r: the number of siblings to the right of the node, including the node; and (iii) d: the number of children of the node. Each path is now a list of 4-tuples - $(name, l, r, d)$. The computation of *score* in line 7 of Algorithm 1 is now replaced with *match_paths* from Algorithm 2. Given an anchor element with ID, a path to an element P_1 computed on the old tree, and a path to an element P_2 computed on the modified tree, *Trackr* first checks the names of all elements

along these paths (line 3). For i^{th} element in P_1 and P_2, if both l and r indices match, then the score is incremented by $\frac{2}{|P_1|}$ (lines 5-6). If only one of indices, say l, matches and the number of children d match, then the score is incremented by $\frac{r(P_1[i])}{|P_1|(l(P_1[i])+r(P_1[i]))}$. Note that this increment is less than the increment when both l and r match i.e. there is a penalty if one of the indices doesn't match. Also note that, *Trackr* uses the number of children as an additional matching criterion in the score computation to discourage any false positives that may arise. Figure 5.8 shows an overview of the path resiliency optimization.

Algorithm 2 Score computation with path resiliency

```
1:  procedure match_paths(P1, P2)
2:      score ← 0, i ← 0
3:      if names(P1) = names(P2) then        ▷ names() returns a list of names of all elements along the path
4:          while i < |P1| do
5:              if l(P1[i]) = l(P2[i])&r(P1[i]) = r(P2[i]) then
6:                  score ← score + 2/|P1|
7:              else if l(P1[i]) ≠ l(P2[i])&r(P1[i]) = r(P2[i])&c(P1[i]) = c(P2[i]) then
8:                  score ← score + l(P1[i])/(|P1|(l(P1[i])+r(P1[i])))
9:              else if l(P1[i]) = l(P2[i])&r(P1[i]) ≠ r(P2[i])&c(P1[i]) = c(P2[i]) then
10:                 score ← score + r(P1[i])/(|P1|(l(P1[i])+r(P1[i])))
11:             else
12:                 return 0
13:             end if
14:             i ← i + 1
15:         end while
16:     end if
17:     return score
18: end procedure
```

With this optimization in place, the fingerprint of a_5 computed from the old tree will now be sufficient to find it in the modified tree, as one of it's index (r) in the path from DIV d_2 and the number of children remain intact.

(ii) $Trackr_{WP}$ **Weighted Path Matching:** Assuming uniform distribution of changes across the DOM tree, longer paths have a higher probability of breaking with time compared to shorter paths (see discussion in Section II-D). Consider a case where in there are two elements with IDs in a tree. Also consider an element whose fingerprint has two paths P_1, and P_2 (from the two elements with ID a_1, and a_2, respectively). In a modified tree, it is

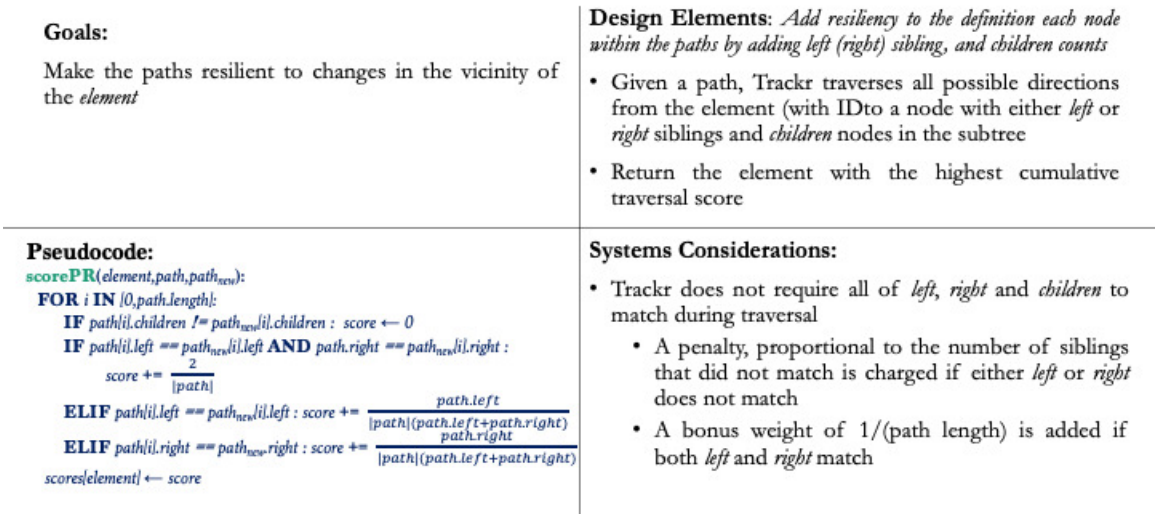


Figure 5.8: Overview of *Trackr* with Path Resiliency

possible that two different elements e_1, and e_2 have the same match score from Algorithm 2. This can occur when the path from a_1 to e_1 matches completely with P_1, and the path from a_2 to e_2 matches completely with P_2. In such a scenario, the probability that the longer path among P_1 and P_2 points to an incorrect element is higher than it's alternate.

Based on this intuition, *Trackr* allocates more importance to matching shorter paths compared to longer paths. This is achieved by multiplying the *score* from Algorithm 2 with a weight that monotonically decreases with an increase in path length[1]. *Trackr* uses an inverse logarithm function $\frac{1}{\ln(1.25+length)}$ to weigh scores[2]. Figure 5.9 shows an overview of *Trackr* with the weighted paths optimization.

(iii) $Trackr_{PL}$ **Path Length Amendment:** To find whether two paths in different fingerprints lead to the same element, *Trackr* first checks if the names of elements along the paths are equal. Consider a case wherein an element is deleted along a path but the rest of the path remains intact. In this case, the names of elements will no longer match. Further, if this deletion is close to the element (say it's parent), it is highly possible that all the paths

[1]When some areas of the DOM tree are subject to more changes than other areas, the assumption on the uniform distribution of changes does not hold. In this case, more weight can be allocated to paths that do not go through change-prone areas. Finding these areas is beyond the scope of this paper

[2]Any monotonically decreasing function will produce the same results

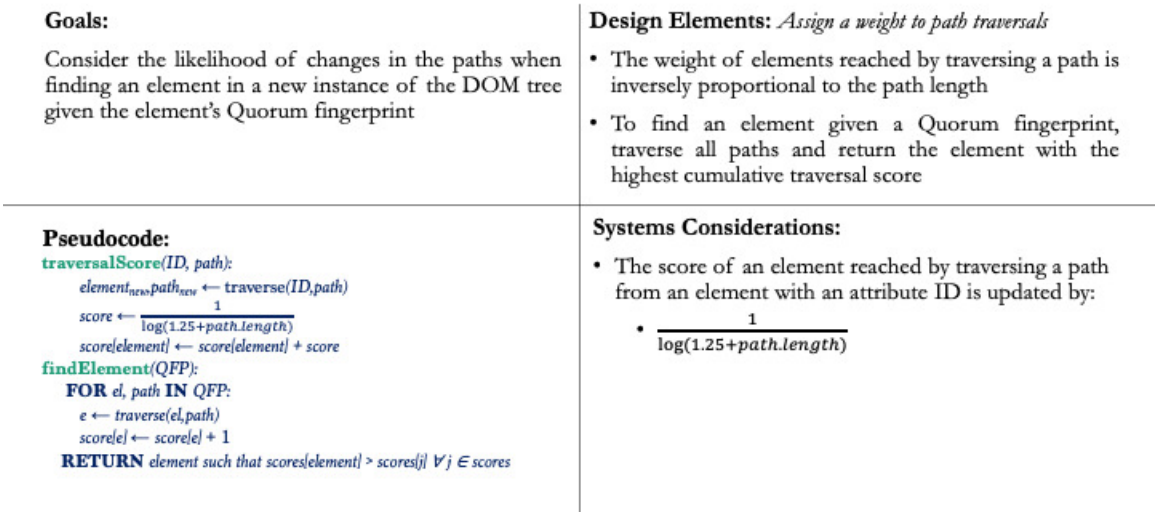


Figure 5.9: Overview of *Trackr* with Weighted Paths

within the element's fingerprint will fail. Through this optimization, *Trackr* accounts for one possible deletion with Algorithm 3. Given a path from a fingerprint computed on the old tree P_1 and a path from a fingerprint computed on the modified tree P_2 (corresponding to the same element with ID as in P_1), if the length of P_2 is one less than that of P_1, *Trackr* creates a set of dummy paths. For every i^{th} element along the path P_1, *Trackr* creates a dummy path P_1' that indicates what P_1 would look like if the i^{th} element was deleted. If the names of elements along this dummy path match to that of P_2, *Trackr* appends this dummy path into a list of candidate paths for consideration (lines 5-12). When an element is deleted, all the element's children are appended to its parent. To account for this, if the original tree has to be traversed 'DOWN' to reach the deleted element (from the previous element in the path), the siblings count l and r of the deleted element are added to the siblings count of the next element along the dummy path. This is because the next element is a child of the deleted element. On the other hand, if the direction of movement is 'UP', the siblings count of the deleted element are added to those of the previous element along the path (lines 13-19). In the end, each candidate path is matched to P_2 using Algorithm 2, and the final *score* is set to the maximum of all scores (among the candidate paths). In addition, a penalty of 10 is added to the length of path P_1 to discourage false positives (lines 22-27).

Algorithm 3 Matching paths with path length amendment

procedure $match\_paths\_weighted(P_1, P_2)$
  $score \leftarrow 0$
  if $names(P_1) == names(P_2)$ **then**
    $score += match\_paths(P_1, P_2) \cdot \frac{1}{\ln(1.25+|P_1|)}$
  else if $|P_2| + 1 == |P_1|$ **then**
    $candidates \leftarrow []$
    $temp \leftarrow P_1$
    for $i \leftarrow 0; i < |P_1|; i++$ **do**
      $temp \leftarrow temp \backslash temp[i]$
      if $names(temp) \neq names(P_2)$ **then**
        continue
      end if
      if $dir(P_1[i]) == `DOWN'$ **then**
        $l(temp[i]) += l(P_1[i+1])$
        $r(temp[i]) += r(P_1[i+1])$
      else
        $l(temp[i-1]) += l(P_1[i-1])$
        $r(temp[i-1]) += r(P_1[i-1])$
      end if
      $candidates.add(temp)$
    end for
    $scores \leftarrow []$
    for $P \in candidates$ **do**
      $scores \leftarrow match\_paths(P, P_1) / \ln(1.25 + |P| + 10)$
    end for
    $score \leftarrow max(scores)$
  end if
  return score
end procedure

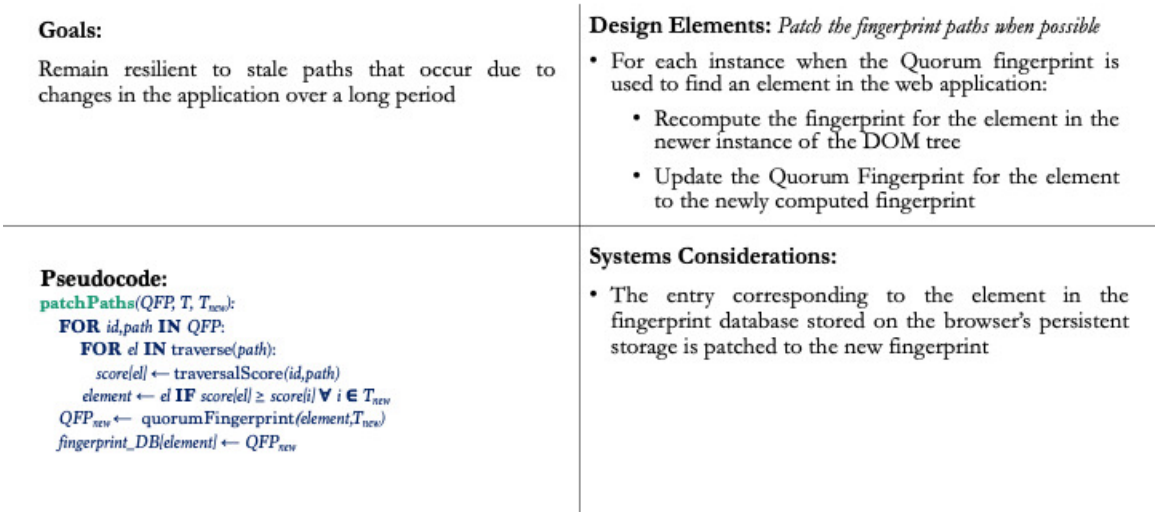


Figure 5.10: Overview of *Trackr* with Progressive Path Patching

(iv) $Trackr_{PP}$ **Progressive Path Patching:** Through baseline quorum fingerprinting and the previous three optimizations, an element can be reliably identified even when the DOM tree is changed. When a web service utilizes *Trackr* to track some elements, their fingerprints are computed and stored. Over time, as the web application undergoes more changes, the paths within the old fingerprint slowly become irrelevant. To avoid this issue, *Trackr* progressively updates the fingerprint every time the user visits the same web application. To do this, *Trackr* first identifies the elements in the web application using the matching procedures outlined earlier. If any of the paths in fingerprint have since been modified, *Trackr* patches the stored fingerprint to reflect the new paths.

(v) $Trackr_{LS}$ **Local Signature:** While all of the previous optimizations are designed to create resiliency in the presence of changes away from the element, when the element itself migrates to a different part of the tree either by itself, or as a part of migration of one of its ancestors all of the paths in the fingerprint can fail. However, there is still a high possibility that the element's surrounding context remains the same (as the element migrates with its descendants). As a fail-safe for this situation *Trackr* includes an element's local context,

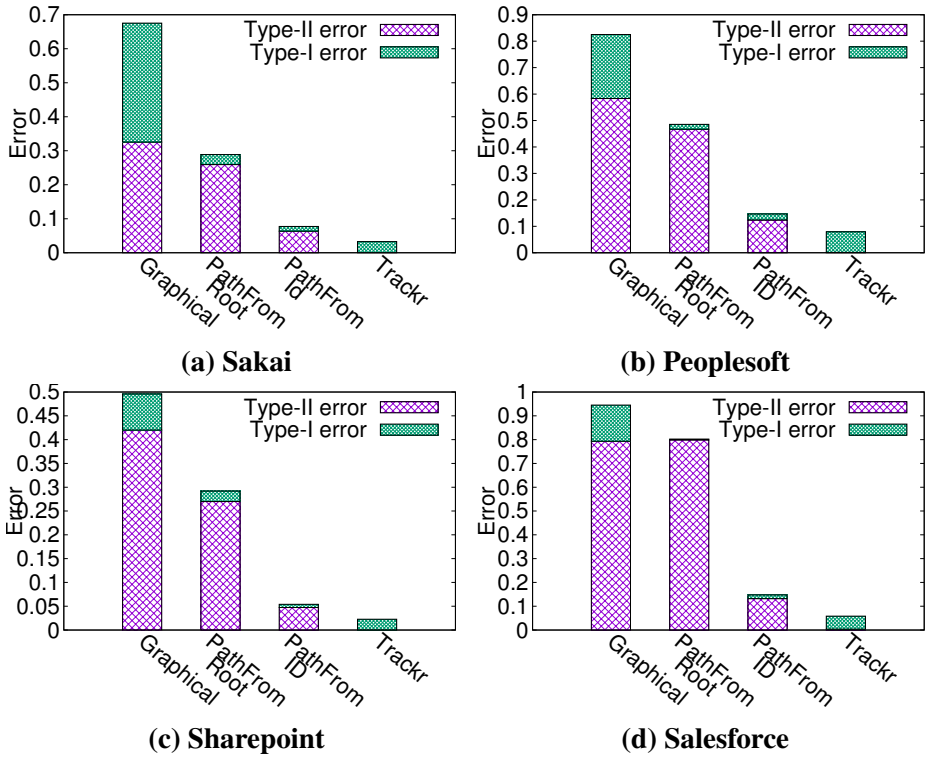


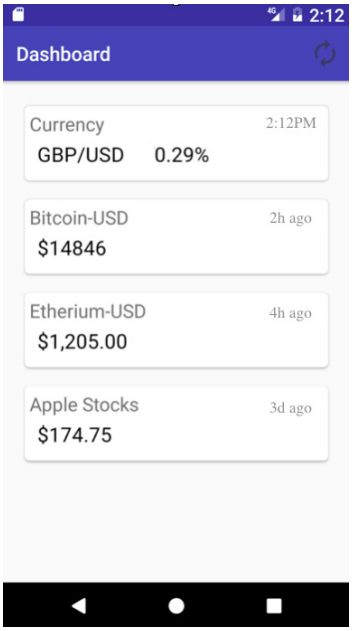
Figure 5.11: Performance of *Trackr* compared to Graphical (Coordinates), Path From Root, and Path From ID

called its *signature* in the fingerprint. The *signature* of an element is defined as a list of tag names of the children and grandchildren of the element ordered in a depth first pattern. To find an element using its *signature*, *Trackr* matches the pattern in *signature* to all other elements in the DOM tree and looks for an exact match. As the signature has a very high rate of false positives, it is only used when the all the paths fail.

5.4 Evaluation

5.4.1 Prototype:

To demonstrate the usage of *Trackr*, we developed a multi-application *Dashboard*, a proof-of-concept mobilization app for Android. Using *Dashboard*, users can mobilize and monitor values within UI elements spanning across multiple web applications within one mobile app. Figure 5.12 shows a screenshot of *Dashboard* app through which four values



Values Tracked
1. GBP-USD gains www.nytimes.com
2. Bitcoin price www.bitcoin.com
3. Etherium price www.coinbase.com
4. Apple stock price www.nasdaq.com

Figure 5.12: Dashboard App

from different websites (or web applications) are tracked. *Dashboard* prototype consists of three components: (i) Configuration chrome extension on the user's computer: To monitor a value through *Dashboard*, the user installs this extension and selects the value to be tracked through a dropdown menu on the UI element containing this value. The extension uses *Trackr* to generate a fingerprint for that element; (ii) Python *Dashboard* server on Amazon EC2: This server uses the fingerprint generated by the chrome extension to track UI elements and periodically monitor the value contained within them. If there is a change in value, the server pushes the new value to the *Dashboard* app (see Figure 5.16) ; and (iii) Android *Dashboard* app on the user's smartphone: This app is responsible for displaying the configured values to the user and updating these values upon a push notification from the server.

Methodology: In this section, we evaluate the performance of *Trackr* on four different web applications: (i) Learning Management - *Sakai[128]*, (ii) Human Resources Management - *Oracle Peoplesoft[148]*, (iii) Collaboration and Team Management - *Microsoft Sharepoint[149]*, and (iv) Customer Relationship Management - *Salesforce[5]*. For each

Table 5.1: Default Experimental Parameters

Name	Value	Name	Value
# of iterations	50	Probability of change	0.5
# of rounds of change	7	# of tracked elements	30

of these websites, we first download the homepage after login and extract the DOM tree. On an average, the number of elements in the DOM tree were 191, 1356, 1357, and 1886, for Sakai, Peoplesoft, Sharepoint, and Salesforce, respectively. We then introduce several rounds of change into this DOM tree. At every change round, each DOM element undergoes a change with a probability p_{change} (default value = 0.5%)[3]. Each change round represents the modifications to the DOM tree between two consecutive visits. The default number of rounds of change is set to 7. At the end of each change round, *Trackr* patches the fingerprint (III-A-iv).

For elements that are selected to change, the type of change is chosen randomly among: (i)Attribute change: The value of a randomly chosen HTML attribute is changed to a new value; (ii) Attribute insertion: A new HTML attribute is added to the element's tag; (iii) Attribute deletion: A randomly chosen attribute is deleted from the element's tag; (iv) Insertion: A new element is inserted as a child of the element at a randomly chosen index. The type of this element is randomly selected among all previously seen tags in that DOM tree; (v) Deletion: The element is deleted from the DOM tree and any children are inserted back into the deleted element's position; (vi) Migration: The element, along with its descendants, are moved to a different (randomly selected) location in the DOM tree; These changes broadly reflect the types of changes an element is subjected to in reality.

At the beginning of every iteration, we download the website, extract the DOM tree and select 30 candidate elements (at random) from the DOM tree to be tracked by *Trackr*. We exclude nodes that only carry meta information such as html, body, head, script, link, meta, etc. We then introduce several rounds of change. After completion of all change rounds,

[3]Even though this probability is small, given the size of a typical DOM tree, the number of changes with each round are high.

we use *Trackr* to find the candidate elements in the modified DOM tree. To establish the ground truth, at the beginning of each iteration, we add a unique dummy ID for each element in the DOM tree. At the end of the iteration, we compare this dummy-id to the dummy-id of the element returned by *Trackr*. We then compute: [4]:

- (i) *Type-I error* = $\frac{\#\ of\ candidates\ wrongly\ identified}{Total\ \#\ of\ candidates}$ (when the dummy ID of the element returned by *Trackr* is not equal to the dummy ID of the candidate);
- (ii) *Type-II error* = $\frac{\#\ of\ candidates\ not\ found}{Total\ \#\ of\ candidates}$ (when *Trackr* is unable to find the element in the DOM tree, but the element was not deleted); (iii) *Error* = *Type-I error* + *Type-II error*;

Type-I errors signify cases where the dummy ID of the element returned by *Trackr* is not equal to the dummy ID of the original candidate element. Type-II errors signify cases when *Trackr* is unable to find the element in the DOM tree, but the element was not deleted during the change rounds. To eliminate random bias, we repeat the experiments for 50 iterations. We also evaluate these errors for three other fingerprint candidates used in prior work: (i) Graphical coordinates (*Graphical Coordinates*), (ii) Path from the root of the DOM tree (*Path From Root*), and (iii) Path from the nearest ancestor with an attribute ID (*Path From ID*).

Macroscopic results:

Figure 5.11 shows the errors of fingerprint candidates using the default parameters from Table 5.1. *Trackr* clearly outperforms all other candidates. On an average, *Trackr* is inaccurate only 4.74% of the time, where as the average error rates for *Graphical Coordinates*, *Path From Root*, and *Path From ID* are 69.74%, 45.44%, and 10.64%, respectively. *Graphical Coordinates* have the highest error rate and it performs worse compared to the fingerprints that rely on the DOM. We can also observe that *Path From ID* has a much lower error compared to *Path From Root*. This improvement can be attributed to the decrease in the path length by computing the path from an element in the vicinity of the given element. This

[4]These errors are not the true definitions of Type-I and Type-II errors used in statistics

Table 5.2: Effect of different optimizations on *Trackr*

Fingerprint	% Error	Fingerprint	% Error
Graphical Coordinates	61.9	Path From Root	22.1
Path From Nearest ID	8.9	*Trackr*-baseline	5.8
Trackr-path_resiliency	4.8	*Trackr*-weighted_matching	4.2
Trackr-progressive_patching	3.9	*Trackr*	2.8

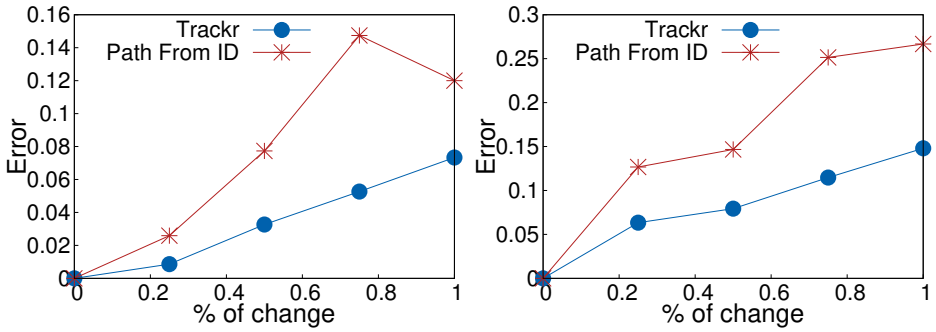
is because the shorter paths have a lower probability of breaking. Through the principle of quorum fingerprinting, *Trackr* achieves an improvement of 55% over *Path From ID*. By building redundancy into the fingerprint by computing paths from many elements, adding resiliency to the paths, giving more importance to shorter paths, accounting for deletions, patching fingerprints when possible, and by using the local signature when all of the above fail, *Trackr* achieves a 55% improvement over *Path From ID*.

Microscopic results:

We also study the improvement resulting from progressively applying optimizations to the baseline algorithm for Sakai application. Table 5.2 shows the overall error percentages as the different optimizations are progressively added to the baseline version (*Trackr*-baseline) of the fingerprint. When we add the path resiliency optimization to baseline quorum fingerprinting, by including two different indices in the paths, the error is reduced from 5.8% to 4.8%. By introducing weights proportional to the path lengths and including the path length amendment optimization, the error is further reduced to 4.2%. By patching the fingerprints on every visit to the web application, the error reduces to 3.9%. Finally, by using local signatures to find the elements when all the paths break, the error rate of *Trackr* is reduced to only 2.8%. These numbers clearly demonstrate the benefits of each of the optimizations.

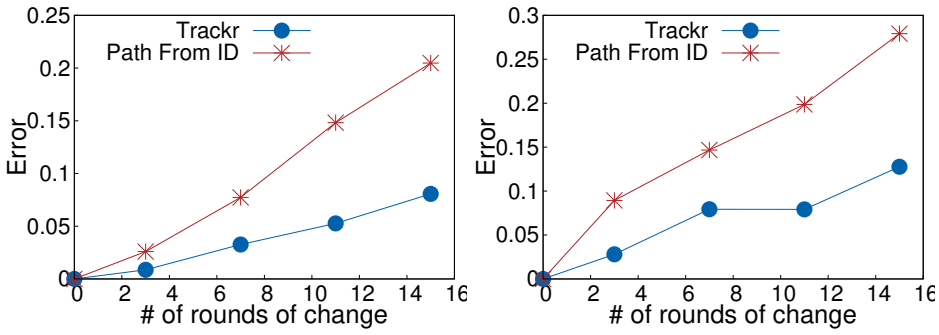
Sensitivity Analysis:

In this section, we study the sensitivity of *Trackr* to different parameters for two web applications - Sakai and Peoplesoft. Unless mentioned, the experiments use the default param-



(a) Sakai **(b) Peoplesoft**

Figure 5.13: Sensitivity to % of nodes changing in DOM



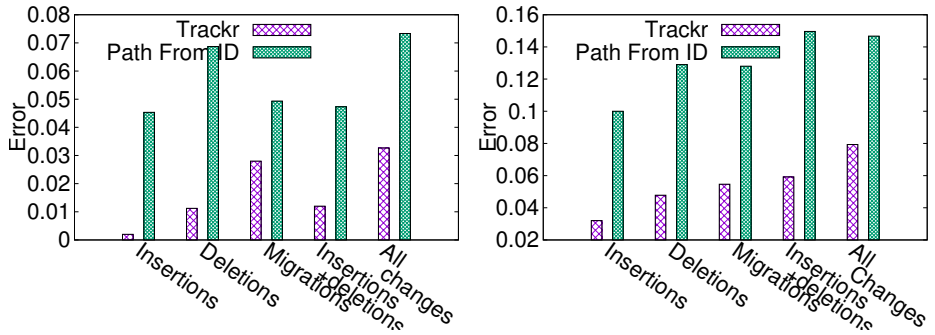
(a) Sakai **(b) Peoplesoft**

Figure 5.14: Sensitivity to the number of rounds of changes

eters from Table 5.1. For relative comparison, we also show the performance of *Path From ID*.

Figure 5.13 shows the effect on the error of changing the percentage of nodes subject to modification in each round of change. As the percentage of change increases, the error rate also increases for both *Path From ID* and *Trackr*. This is because as the DOM undergoes more changes, the chances of the paths in the fingerprint breaking also increase. The increase in error is roughly linear. On an average, every 0.1% increase in probability of change results in a 1.1% increase in error for *Trackr*, and a 1.9% increase for *Path From ID*.

We also study the effect of the type of change on the error (figure 5.15). With only insertions allowed, the average error rate is 1.7% for *Trackr* and 7.3% for *Path From ID*. When only deletions are allowed, the average error rate is 2.9% for *Trackr* and 9.9% for



(a) Sakai **(b) Peoplesoft**

Figure 5.15: Sensitivity to types of changes

Path From ID. For only migrations, the average error rate is 4.1% for *Trackr* and 8.8% for *Path From ID*. When deletions are also allowed, the average error rate increases to 3.56% for *Trackr* and 9.85% for *Path From ID*. When all the different types of changes are allowed, the average error rate is 5.6% for *Trackr* and 11.3% for *Path from ID*. As different types of changes are introduced, the error rate increases. Given that the probability of change is the same, if all changes are equal, the error should remain the same. However, these numbers indicate that *Trackr* is most resilient to insertions and most sensitive to migrations. This is because when a node migrates, all the paths in the fingerprint are broken leaving *Trackr* with only local signature to find a match. However, since the local signature is more susceptible to finding the wrong elements, the error rate for migrations is higher.

Figure 5.14 shows the effect of changing the number of rounds of change, the DOM tree is subject to, before the error is computed. As the number of rounds are increased from 0 to 15, the error rate also increases. On an average, per round of change, the increase in error rate is about 0.7% for *Trackr* and 1.63% for *Path From ID*. The increase in error rate is lower for *Trackr*, as the paths in the fingerprint are progressively updated after every round of change. While this number seems alarming, in reality, DOM trees change very slowly with time, and hence *Trackr* still remains robust for a long period of time.

5.5 Use Cases

In this section we discuss how *Trackr*'s accurate and reliable fingerprinting of UI elements can be integrated with the three secondary service use cases we introduced in Section 5.2.

5.5.1 Automation:

Web automation services are services that allow a user to perform a set of actions on web applications programmatically. They are particularly useful when a sequence of actions has to be repeatedly executed many times. For example, for large-scale testing of web applications, the testers have to perform the same action (like typing text into a textbox) many times with different parameters (say different values of text). Using automation services can help relieve this task burden. Selenium [145] is an example of such a browser automation framework. To automate a workflow with Selenium, the developer has first to obtain a handle for the UI element using a script with DOM access methods and specify the type of action with any required parameters in a script. Selenium interprets the automation script and performs the specified actions on a browser as if they were executed manually. For example, to enter text in a text box, the developer has to script how to access the text box element, say through xpath expressions, and use the method *find_element_by_xpath()* to obtain a handle. Text can be inserted by calling the method *send_keys()* on the handle. The burden of obtaining the right handle for an element rests with the developer. If the web application changes after the automation scripts have been written, Selenium will not be able to perform the specified actions on the desired element. The developer then has to manually update the scripts with methods to access the correct handle for the elements within the modified DOM tree. For web applications that frequently change, this is burdensome and impractical.

Trackr can alleviate the problem of re-coding handles for elements every time the application changes, by allowing the developers to create a robust fingerprint for the elements.

By including the *Trackr* browser extension through $add\_extension()$ method of selenium, and calling $Tracker.track()$ on the element's current handle, a tracker for the element can be initialized. At a later point in time, the correct handle to the element can be obtained by passing the tracker's name to $Tracker.find()$ method. The following pseudocode demonstrates the usage of *Trackr* in Selenium.

```
\\Adding trackr extension
options = webdriver.ChromeOptions()
options.add_extension('trackr.crx')
driver = webdriver.Chrome(chrome_options=options)
...
\\ Track a list of elements
\\ elem: a handle for the element to be tracked.
\\ name: a name for the tracker
js='return Trackr.track(arguments);'
fp = driver.execute_script('js',elem, name)
...
\\ Get a handle for tracked element
js = 'return Trackr.find(arguments)'
elem = driver.execute_script('js',name)
...
```

Macro-Creation:

There are a wide variety of web applications available today that provide a diverse range of services to the end user. The types of services include, but are not limited to, productivity (e.g., Microsoft One Drive, Google Drive, Microsoft Sharepoint, etc.), home automation (e.g., Nest, Hue, etc.), personal assistants (e.g., Alexa, Google Home, etc.), Collaboration management (e.g., Microsoft Sharepoint), HR and Customer management (e.g., Oracle

Peoplesoft, Salesforce CRM, etc.), travel management (e.g., Kayak, Concur, etc.), weather and traffic (e.g., maps, Weather.com, etc.), fitness (Fitbit, runkeeper, etc.) etc. Even though different web applications provide services that pervade the day to day life of users, most of these applications exist independent of each other. Macro-creation services are secondary services that allow users to create 'macros' to conveniently access the services provided by web applications on a standard interface. IFTTT (If This Then That) [146] is a popular macro-creation service using which users can monitor some variables within one service, create triggers when these variables meet some conditions, and perform actions on a different web service when the triggers are activated. Using IFTTT, users can monitor parameters from one web service and create triggers on these parameters based on some conditions, and perform actions on a different web service when the triggers are met. For example, users can create a trigger to monitor the current temperature on weather.com and set the thermostat of the house to cool when it crosses a threshold. In IFTTT, macros can be configured through a GUI, wherein the users can select from a list of available triggers and actions. The burden of providing the triggers for IFTTT is on the web application and therefore, the users are restricted to only those applications that expose an IFTTT compatible API. However, given that a very small percentage of applications provide an API, the benefits of macros are severely limited. On the other hand, expecting all web services to provide a functional API to monitor variables and perform tasks is impractical. Further, the burden of providing the right handles to monitor parameters (say the outside temperature) rests with the primary web service. If the primary web service fails to maintain these handles, the applets created by users can fail.

With simple extensions to *Trackr*, users can be allowed to create their triggers even from web applications that do not currently provide an API to integrate with IFTTT. Rather than manually updating the API to monitor a trigger value each time the structure of the primary web service change, the developers (of primary web services) can use *Trackr* to track web elements that contain the trigger value automatically. Further, using *Trackr*, users

can create their triggers from web services that do not currently provide an API to integrate with IFTTT. To support this feature, *Trackr* browser extension can be extended to allow users to select a web element to be monitored by right-clicking on it in a web page and selecting an option from the context menu. The users can then be asked also to enter a condition for the monitored value. *Trackr* can then periodically monitors the element from the application. When the trigger conditions are met, *Trackr* can embed the value in a JSON object and send a response back to IFTTT as an HTTP POST message indicating that the trigger is activated. These steps are demonstrated in the following pseudocode.

```
elem, condition <- get from user
// Start a tracker to monitor element
fp = trackr.Track(elem,name);

//periodically monitor value of elem
while(1):
 load_web_service() // Load the web service
 // Get the monitored value
 value = Trackr.find(name).value
 if value satisfies condition:
  //Create a IFTTT JSON response
  response_json= {
   "trigger_identity": "92429d82a41e93048",
   "triggerFields": {"monitored_value": value},
   "ifttt_source":
    {"url": "https://example.com/trigger"},
  }
  // post the response back to IFTTT
  post(response_json);
```

```
sleep(period) // polling frequency
```

Application Mobilization:

Application mobilization is the process of transforming desktop applications into smartphone apps. It can improve the productivity of enterprise employees, as they can now perform tasks that were originally performed on a desktop, but their smartphone. Among different strategies to mobilize enterprise applications, application refactoring is gaining popularity today. Mobilizing enterprise applications with refactoring involves hosting the application on a cloud server and providing a highly optimized native UI for the users to interact with the application on their smartphones. Here, the core application logic remains unchanged and only the front-end is transformed into a highly optimized native UI for the smartphone. However, unlike traditional remote computing that presents the user with the application as-is, the UI is instead transformed into a platform native UI. A crucial step in this process involves capturing any actions performed by the user on the smartphone native UI and executing them back on the original UI of the application. As refactoring does not require modifying the original application in any way, mobilization can be readily achieved with minimal development and deployment costs. Through Capriza [147], users can create micro-apps that perform specific workflows on traditional enterprise applications through a simple GUI tool called the Designer. It allows the users to select elements from the original application UI, customize them and add them to the smartphone UI. For these selected elements, Capriza creates unique fingerprints and associates them to their smartphone native versions. When the user performs an action (say taps a button on the smartphone), these mappings are used to find the corresponding UI element of the original application and execute actions. For the created mobile app to function correctly, the actions have to be executed on the correct elements in the original UI. When the layout of the original application changes, it is possible that the fingerprints generated by Capriza at the time of micro-app creation fail. In this case, the user will not be able to perform the intended work-

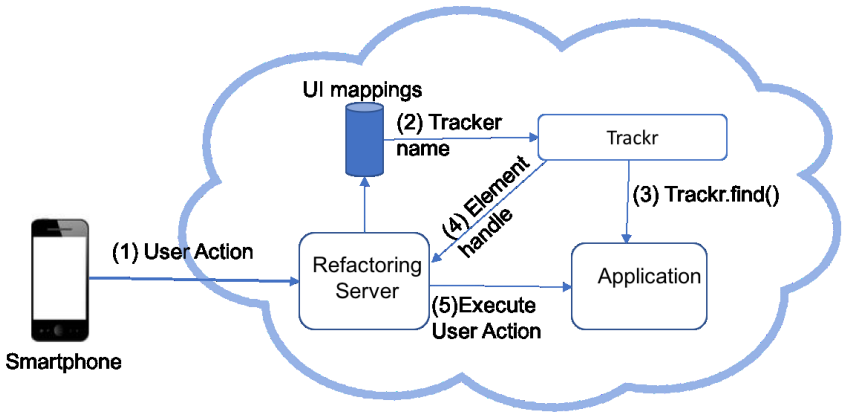


Figure 5.16: Integration of *Trackr* with a mobilization service

flow on the micro-app. The user will now have to recreate the original micro-app from the modified application UI.

Figure 5.16 demonstrates the possible architecture that integrates *Trackr* with Capriza. When the user selects the elements from the Designer, Capriza can use *Trackr* to initiate trackers for them and map these trackers to their corresponding native UI elements. When the user performs an action on the native UI, the tracker names can be used to obtain a handle to the element in the original UI and perform the corresponding action on it.

5.6 Issues

The following questions could be raised on the approach taken by *Trackr* to track elements:

- *Can Object tracking algorithms from image processing research [150] be used to track elements?* Object tracking algorithms assume that between two consecutive video frames, the object does not move by a lot. However, given that the web elements are containers of content, their appearance can change drastically between two instances. Therefore, pixel-based object tracking methods do not apply to our problem;

- *Can the developers of web applications be forced to declare attribute IDs for all web elements?* It requires remodeling the large body of legacy web applications and is

impractical;

- *Can trackers be embedded within elements by the secondary web services?* This would require a change at the web application's end to honor the trackers, and is therefore impractical;

CHAPTER 6

PEEK: A MOBILE-TO-MOBILE REMOTE COMPUTING PROTOCOL

6.1 Introduction

The adoption of smartphones (and tablets[1]) has seen an explosive growth over the last decade and in 2011 the number of smartphones shipped finally eclipsed that of the number of PCs. Even traditionally conservative enterprise sector is adopting mobile devices at a blistering pace, driven by a clear return-on-investment in the form of higher employee productivity, reduced paper work, and increased revenue. It appears inevitable that smartphones will become the primary computing device for a majority of users in the future.

Many enterprise employees work in teams where collaboration between various team members is necessary to accomplish tasks [151]. Collaboration between enterprise employees has been shown to improve employee productivity [151]. Even though collaboration is crucial in an enterprise setting, very few mobile apps (e.g., Google Docs) natively support it. Most mobile apps are designed for individual use and do not allow multiple users to collaborate unless they are working on the same device. Such a restriction hinders one of the significant advantages of mobility - the convenience of working from any location. In this chapter, we consider using mobile-to-mobile remote computing to enable collaboration between users on two different devices in scenarios when the application does not include simultaneous multi-user support. Remote computing involves a remote server running applications on one user's mobile device while the other user interacts with it remotely, using a remote computing protocol. Remote computing allows users to view and control other devices in real time while being physically away from them.

In addition to collaboration on smartphone apps, a mobile-to-mobile remote computing

[1]While all of our discussions apply to both smartphones and tablets, for brevity we refer only to smartphones in the rest of this paper.

protocol can enable users to experience a range of different application scenarios, which otherwise would not have been possible. For example, a user can allow her colleague access to her smartphone to get help with editing an image. She can play a game on her friend's smartphone, even when she is present at a different physical location. In addition to that, she can create virtual smartphone images on a resource-rich cloud infrastructure and remotely access them to perform CPU heavy tasks. She can also help configure her grandmother's phone by controlling it remotely. The possible applications with a mobile-to-mobile remote computing protocol are hence numerous.

While several remote desktop sharing protocols are available today [126, 152–154], they cannot be applied as-is for mobile-to-mobile remote computing for the following reasons: (i) Multi-touch interface: Existing protocols assume that the user interacts with her device using a keyboard and mouse. However, most smartphones use multi-touch screens, which are not supported by these protocols; (ii) Context association: A user interacts with her smartphone, not just through the input devices, but also with the associated context through sensors (e.g., accelerometer, proximity sensor, gyroscope, light sensors, location sensors) for a rich application experience. However, traditional remote computing protocols do not associate context to a session; (iii) Resource constraints: A good-quality remote computing session requires high network bandwidths and substantial processing capabilities. While the resource requirements are available within most Desktops, smartphones are limited by low power processors and limited bandwidth wireless networks (WiFi, 3G/4G).

In this context, we introduce *Peek*, an application agnostic, platform and device independent mobile-to-mobile remote computing protocol for smartphones. *Peek* has the following properties: (i) Multi-touch support: *Peek* enables client-server interaction through multi-touch interfaces, which increases the ease of interaction. Compared to Virtual Network Computing (VNC), a popular remote desktop solution, *Peek* vastly increases the number of supported touch gestures. By implementing *Peek* on Android smartphones, we show that the time taken to perform specific actions on the server remotely is reduced by 62.8%

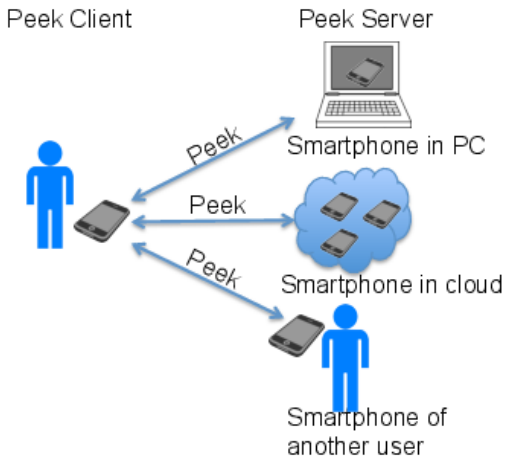


Figure 6.1: *Peek* usage

on average; (ii) Context association: *Peek* associates sensor context to a session, which allows users to experience a broad range of smartphone apps (including apps using local device context) remotely; (iii) Multi-modal frame compression: *Peek* chooses a frame compression mode based on the server's CPU/memory load, the rate of change of screen pixels and the current network bandwidth. Using synthetic datasets, we show that *Peek* can potentially reduce the bytes sent over a network by over 30% compared to VNC. To the best of our knowledge, *Peek* is the first ever remote computing protocol designed for communication between two smartphones. In the rest of this chapter, Section 6.2.1 provides a primer on remote computing.

The rest of this chapter is organized as follows - Section 6.2.2 outlines the need for a mobile-to-mobile remote computing protocol for smartphones. Section 6.2.3 discusses the key challenges of using existing remote computing protocols for smartphones. Section 6.3 sketches the details of *Peek*. Finally, Section 6.4 specifies the evaluation of *Peek*.

6.2 Background and Motivation

6.2.1 A Primer:

Remote computing involves one or more client devices communicating with a server. During a remote computing session, the server encodes the content of its frame buffer (screen pixels) and sends it to the associated clients. The clients display this view on their screens and allow users to interact with it using input devices like keyboard and mouse. The clients capture these input device operations and send them over the network to the server, which executes these operations at its end and sends any screen updates back to the clients. These updates could either be a direct encoding of the screen pixels[96] or primitives such as 'draw a rectangle'[95]. The format of the messages exchanged between the client and the server depend on the remote computing protocol. VNC uses Remote Frame Buffer Protocol (RFB) [96] for communication. In RFB, server encodes pixels in the frame buffer using a compression scheme negotiated between the client and server at the start of the session. The client decodes these pixels and displays them by writing onto the local frame buffer. Irrespective of the type of encoding, only those rectangles that have changed from the previous state of the frame buffer are sent over the network. The client can control the server through input devices like a keyboard or mouse.

6.2.2 A case for mobile-mobile remote computing

We consider mobile-to-mobile remote computing as a platform that extends a smartphone to a new dimension of applications. With remote computing, users can experience applications through other physical or virtual devices and are not limited by their device. We envision the following applications for mobile-to-mobile remote computing:

Real time collaboration:

Users can collaborate on any smartphone application, even when it is not built for collaboration. One user can run the application natively on her smartphone, while the other uses can use remote computing to access the same instance of the application on using their respective smartphones. For example, user Bob can help user Alice edit a picture on Alice's smartphone by remotely accessing Alice's smartphone from his smartphone and interacting with Alice's instance of a photo editing application.

Computation offload:

A user with a low-end smartphone can access a virtual instance of a device hosted on a resource-rich cloud and complete resource-heavy tasks like panorama stitching, image manipulation, video editing, compression, encryption, among others.

Troubleshooting:

A support technician can access a user's smartphone and help debug an issue in real-time. Such a feature would also enable non-savvy users to get help when needed to utilize the full range of features available with smartphones.

Multi-player gaming:

Games, with or without multi player support, can be enjoyed by multiple users without being present at the same location. For example, by remotely accessing the same smartphone, two users can play Angry Birds, where they can either collaborate to pass a level or take turns competing on the same level and compare scores.

Virtual Mobile Infrastructure:

For data security purposes, certain enterprise workers are required to carry a smartphone for office use in addition to their personal smartphones. This can be avoided if the enterprises

Table 6.1: Touch to mouse translation

Client Gesture	Translation	Server execution
Tap	Left Click	Tap
Double Tap	Left Double Click	Double Tap
Long Press	Long click	Long press
Long press(short) + swipe	Left click + Move	Swipe/Drag

can provide a sandboxed virtual smartphone environment on a cloud, which the employees are allowed to access only at the workplace or with a secure VPN.

6.2.3 Key Challenges

Desktop remote computing protocols[2] cannot be applied as-is for mobile-to-mobile remote computing for the following reasons. In this section, we outline these challenges.

Input Handling:

Desktop remote computing assumes one end of the application is a Desktop with which users interact with a keyboard and a mouse. With the advent of smartphones, interactions through keyboard and mouse are no longer relevant as users interact with applications on a smartphone using touch screen gestures. However, there are no remote computing protocols specifically designed for touch screen input devices. Existing VNC smartphone client and server applications are designed for keyboard and mouse operations. They adapt to the desktop protocol by translating touch screen operations into mouse operations, rather than supporting them natively. A VNC smartphone client that translates touch to mouse operations can be used to communicate with a VNC smartphone server that converts the mouse operations back to touch operations. We identified this translation between touch and mouse operations and present it in Table 6.1. Such a translation creates the following problems:

(i) *Many gestures cannot be mapped*. Multi-touch enables a smartphone user to interact

[2]While we use VNC as a representative desktop remote computing protocol, the discussion can still be applied to all the other protocols.

Table 6.2: Non intuitive and non existent gestures

Swipe	Scroll	Multi-finger Swipe	Multi-finger hold
Multi-finger drag	Pinch	Multi-finger tap	Multi-finger pinch
Expand	Multi-finger expand	Multi-finger multi-tap	Fling
Multi-finger Fling	Multi-finger rotate	Anchoring	

with her device using intuitive gestures performed with multiple points of contact. Since a mouse has only one pointer moving across the screen, many multi-touch operations cannot be mapped;

(ii) *Mapped gestures are not intuitive*. For example, when a user wants to scroll a list on his smartphone, she swipes upwards on the touch screen. However, if the user wants to do the same on the VNC server smartphone, she has to long press on the screen first and then swipe up. This usage is confusing to remember as it is not natural and hence is a source of confusion. Also, if the user doesn't swipe on time after she long presses, a menu might open up corresponding to the long press gesture. Gestures that are either non-intuitive or cannot be mapped are shown in Table 6.2.

(iii) *Context information is not associated.* A typical smartphone is equipped with many sensors including gyroscope, accelerometer, magnetometer, proximity sensor etc., which provide contextual information on the device. This sensor context is used by many applications to provide a rich experience to users. For example, many racing games make use of the gyroscope and accelerometer readings to emulate the effects of steering, i.e. if the user tilts to the left, the vehicle is steered to the left. We argue that *apart from input devices like touch screen, mouse and keyboard, sensors are also a way to interact with the device.* Current remote computing protocols only consider input devices like a mouse or keyboard as a way of interacting with the device. Applications requiring client's sensors will not be able to operate with traditional remote computing.

Challenge 1: How does a remote computing smartphone client interact with the smartphone server using multi-touch operations and associate its sensor context to a session?

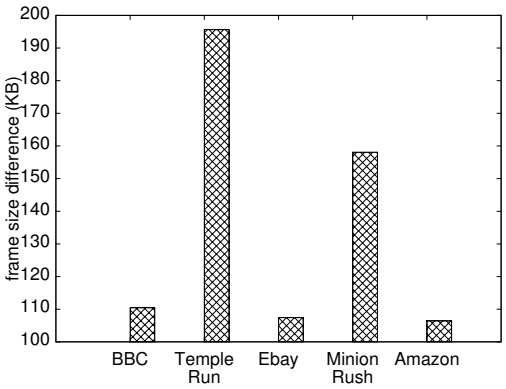


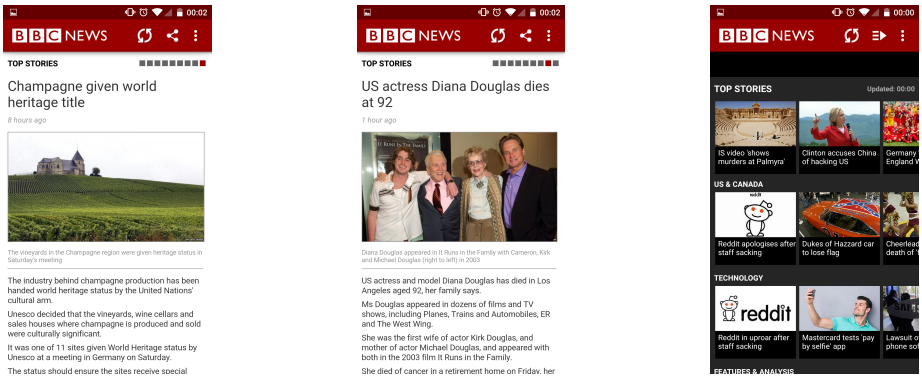
Figure 6.2: Average consecutive frame size difference of different applications

Table 6.3: VNC compression on smartphones

Type	fps	CPU (%)	Memory (MB)	Bytes per frame
Tight 256 Color	22	48	28	13946
ZRLE	5	21	17	42101

Remote View Sharing:

Usage of frame compression techniques designed for desktops, for smartphone-to-smartphone remote computing, results in poor resource utilization in smartphones. Using VNC as-is for smartphones presents the following problems: (i) *Frame compression in VNC is independent of device status.* Compression schemes that have a low CPU overhead have poor , and those that have better visual performance are CPU heavy. To demonstrate this, we setup a remote computing session between a VNC server that generates frames at 60 frames per second (fps) and a client that measures the rate of display of these frames, on two LG Nexus 5 smartphones. Table III shows the fps at client and CPU, memory, bytes per frame sent at the server for two popular VNC compression schemes. We found that 'Tight' achieves better fps (22 vs 5) and better per-frame compression than 'ZRLE', at the cost of higher CPU and memory usage. Using the same frame compression scheme throughout the session irrespective of the device status is not suitable for smartphones, as resources are limited; For example, if the server's CPU utilization is high and network utilization is low,



(a) Article A **(b) Article B** **(c) Home Page H**

Figure 6.3: Different screen layouts of BBC

data could be sent over the network with a simple compression scheme like ZRLE, even though a more complex scheme was fixed beforehand.

(ii) *Applications have different compression requirements.* The rate of change of screen content (frame rate) varies among different applications available. Fig. 6.2 shows the average size difference between consecutive frames, in a single session for different applications on Android. The frame rate for graphically intensive games like Temple Run and Minion Rush is high compared to the other applications. VNC uses the same type of compression scheme, irrespective of the application. Using a complex compression algorithm, which was decided at the start of session for applications with a lower frame rate, leads to wastage of resources;

(iii) *Application usage behavior leads to inefficient resource utilization in smartphones.* Many smartphone applications use a fixed number of screen layouts with different content. For example, in the BBC mobile app, the articles share the same layout, which is different from that of the home screen. Consider a case in which the user opens article A from the home screen H and switches back to home screen, opens another article B. Since a VNC server transmits the changes from the last displayed frame, the size of updates sent are U = diff(H,A) + diff(A,H) + diff(H,B), where diff(x,y) is the size of the screen update if the screen changes from x to y. However, VNC does not leverage the fact that the current frame might have been displayed (H) in the past, or that a similar frame might have been accessed

before (A and B). This results in a wastage of network resources. If the client and server can remember those frames that could repeat (like H) or one representative frame that is similar to many frames that could be displayed in the future (say A or B), the number of bytes sent over the network could be significantly reduced (update size = diff(H,A) + diff(A,B)$<<$U).

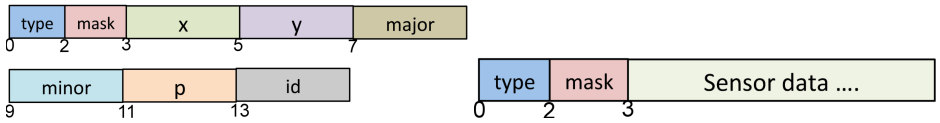
Challenge 2: How can the remote computing server compress its screen, taking into account (i) CPU, memory and network loads, (ii) screen redundancy and (iii) varying frame rate?

6.3 PEEK: A mobile-to-mobile remote computing protocol

In this section, we briefly present *Peek*, a mobile-to-mobile remote computing protocol for collaboration *Peek* is built on the RFB protocol and adds multi-touch support and context association to it. *Peek* also improves upon frame compression of RFB, by using a multi-modal compression scheme. *Peek* deals with the challenges described in Section 6.2.3 as follows:

6.3.1 Multi-touch Support and Context Association:

If a user has to access a smartphone using VNC remotely, she has to use a client application on her smartphone that maps touch operations to mouse operations, and a server application on the remote smartphone that translates mouse operations to touch operations. The mapping between touch and mouse operations affects the usability of the application. *Peek* clients instead, directly capture touch interactions, represent them in a suitable format to avoid loss of integrity, and send them to the *Peek* server for execution. By removing the layer of mouse translation, touch interactions can be natively represented at the client and easily interpreted at the server. This enables the users to interact with the remote server intuitively in the same way as they would interact locally with their smartphones. *Peek* adds a new touch screen input method to the RFB protocol. *Peek* clients represent each



(a) Touch message **(b) Sensor message**

Figure 6.4: Message format in *Peek*

point of contact of user's finger to the screen with a touch message. Each touch message is 14 bytes long. The first byte, *type*, is a constant (=12) for all touch messages, irrespective of the device. It serves as an indication to the server to interpret the next 13 bytes of the stream as a touch message. The second byte *mask* is a bitwise mask that represents the validity of different fields in the rest of the message. In *Peek*, touch contacts are assumed to be elliptical in shape[3] and each touch contact is represented by: (i) position on the screen - x,y (horizontal and vertical coordinates of the center of contact); (ii) dimensions - $major$, *minor* (lengths of major and minor axes); (iii) pressure of contact - p; (iv) Id of the point of contact - *id*. While parameters x, y, $major$, *minor* and p are designed to represent the physical aspects of contact, *id* is useful in a multi-touch scenario to differentiate one point of contact from the other. These parameters are captured in real time by *Peek* clients. The presence of x, y, $major$, *minor*, p and *id* in the touch message is indicated by setting bits 1 to 6 of *mask*, respectively. While simple actions like tap have only one point of contact, other actions (swipe, drag, scroll, etc) have multiple points of contact along the path a finger traces on the screen. Each such touch contact is packed into a touch message. A special message with a mask of '0' is sent to signal the end of an action and is generated when the point of contact leaves the touch screen. When there are multiple points of contact for a touch gesture, some of the parameters might remain the same for these contacts (e.g., p). These parameters can be skipped in subsequent messages, and the mask is set appropriately. The *Peek* server extracts touch parameters from the message and virtually applies the touch contact. If the *mask* indicates that a parameter is not present, the last known value is used.

[3]Most of the touch sensor drivers assume the area of contact is an ellipse.

Peek clients also capture various sensor readings and send them to the server. A user with a *Peek* client has an option to chose either her own device's context or the server's context during a session. *type* value varies with the function of the sensor (15 for the gyroscope, 16 for the accelerometer, 17 for the proximity sensor, etc.). Similar to *mask* in a touch message, *mask* in the sensor message is a bitwise mask that represents the validity of the sensor data. The exact format of representation of sensor readings in a sensor message depends on the type of the sensor. For example, for a gyroscope (accelerometer), it is a series of three double values, representing the rate of device's rotation (acceleration) along X, Y, and Z-axes. For a proximity sensor, it is a binary value, representing if the phone is near/away.

All the major smartphone operating systems provide APIs to interpret touch/sensor activity (e.g. UIApplication class in iOS, and /dev/input/event virtual file system in Android). Also, the implementation of extraction and execution of touch/sensor messages depends on the OS of the device. For a Linux based OS, this can be achieved by writing a series of bytes into /dev/input/event virtual filesystem in a suitable format. Since message format is independent of the OS, clients and servers on different OS can communicate with each other. With this message representation, all possible multi-touch and sensor events can now be captured and communicated, thereby increasing the ease of interaction for users.

6.3.2 Multi-modal Compression:

Peek introduces a new multi-modal frame compression technique that takes into account content redundancy, the rate of change of application content and the device resource usage. The *Peek* server identifies specific key frames from the past session history and compresses the difference between the current frame and a key frame closest to the current frame. In this way, *Peek* reduces the amount of data to be compressed, thereby reducing the amount of data sent over the network and CPU cycles. Also, *Peek* uses video compression techniques when it detects rapidly changing screen content.

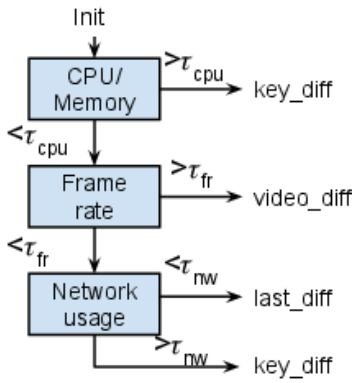


Figure 6.5: Multi-modal compression

While a VNC server uses a compression scheme selected at the beginning of the session, *Peek* server selects one of three compression modes by periodically monitoring its CPU/memory load, network load and frame generation rate: (a) ***last_diff:*** Like in RFB, the difference between the last frame and the current frame is compressed; (b) ***key_diff:*** To save bytes sent on the network in a scenario where the current frame could be very similar to content in the past, the *Peek* server identifies some representative (key) frames in the session history. If the current frame is similar to any one of these key frames, the difference between the key frame and the current frame is compressed and sent along with the index of the key frame. *Peek* uses clustering techniques to identify these key frames. Frames from the session history between a particular server and client, that are similar to each other are clustered into groups using fast online integer K-means clustering algorithm[4]. For each cluster, a frame with the lowest possible difference with the centroid of that cluster is considered as a cluster head. The number of clusters to be formed is chosen based on the current memory utilization of the server and client. Periodically, cluster heads are communicated to the client and are stored as key frames in the memory of both the client and server. This overhead is negligible because cluster heads only need to be communicated infrequently. Without changing the compression algorithm, *key_diff* reduces the burden on the device's resources by reducing the amount of data to be compressed; (c) ***video_diff:*** In

[4]While *Peek* uses K-means, it is one among a broad set of fast and light online clustering algorithms that could be used potentially

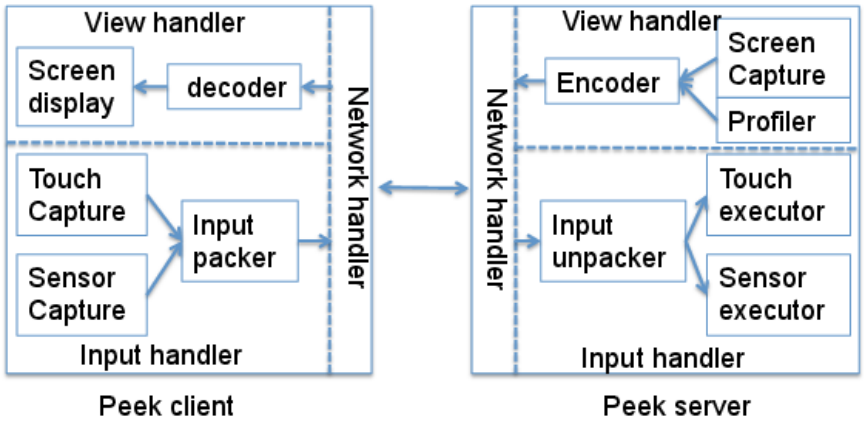


Figure 6.6: System architecture of *Peek*

this scheme, the session is treated as a motion video, and MPEG4 compression is used on it. This compression scheme is mainly designed for graphically intensive applications like games, which have rapidly changing frames. For these applications, a user is presented with new content that is quickly generated through dedicated GPUs. Using *key_diff* that relies on session history does not make sense for these applications as the content is not repetitive. *Peek* utilizes motion prediction and motion compensation algorithms provided in the MPEG4 standard to compress these frames.

Peek server continuously monitors the device and chooses one among the three compression modes based on Figure 6.5. Since CPU/memory is the most important resource that affects a device's usability, not just for remote computing, but for all other applications, *Peek* first considers the CPU/memory utilization to select a mode, and chooses *key_diff* if it is beyond a threshold τ_{cpu}. Otherwise, if frame rate is greater than a threshold τ_{fr}, *video_diff* is used. If frame rate is less than τ_{fr}, *last_diff* or *key_diff* is chosen depending on whether network utilization is less or greater than a threshold τ_{nw}.

6.3.3 System Architecture

Devices running *Peek* have three components: (i) Input handler, (ii) View handler and (iii) Network handler. The functions of these handlers change depending on whether the device is running in the server mode or the client mode (Fig. 6.6). The client input handler

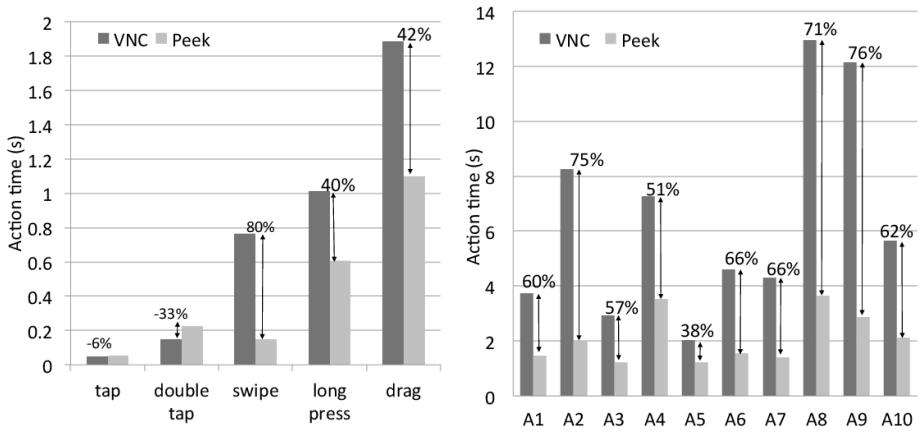
Table 6.4: Action descriptions

Action	Description
A1	Crop a picture using Photo Editor Pro
A2	Write 'A', 'B' and 'C' with finger
A3	Play 6 moves in Candy Crush
A4	Find an email in a list and delete it
A5	Open Youtube and search for 'apple'
A6	Select a paragraph in an email
A7	Find phone's IMEI number from settings
A8	Open a document and append text to the end
A9	Draw a 3x3 grid on screen
A10	Draw a smiley face on the screen

captures all the touch events and sensor events through the touch capture and sensor capture module, respectively. These modules pass the event information to input packer, which packages it into messages. The input unpacker module in server input handler unpacks the messages, interprets the parameters and sends touch/sensor parameters to Touch/Sensor executor which executes them. Server view handler uses the frame capture module to capture the device's screen. Frame encoder chooses the right compression technique for a particular frame based on inputs from the profiler on CPU/memory, network and frame rate, and compresses the frame. The profiler profiles the CPU, memory, network, and frame rate. At the client view handler, the frame decoder decodes the frames, adjusts the image resolution and displays it on the client screen. The client/server network handler is responsible for communication between server and client over the network.

6.4 Evaluation

We implemented and evaluated *Peek* server and client on two LG Nexus 5 smartphones with Android v4.4.4. We build *Peek* on Android VNC Viewer, an open source VNC client and DroidVNCServer, an open source VNC server to handle touch and sensor messages. We extract these parameters from /dev/input/event virtual file system since Android has a Linux kernel. The two smartphones are connected to the same WiFi AP.



(a) Primitive Actions **(b) Complex Actions**

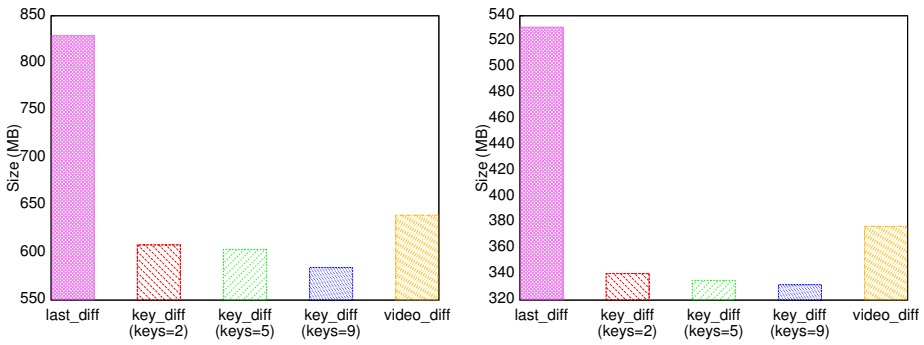
Figure 6.7: Action times

We evaluate the usability of *Peek* by performing specific actions through the client on the server, and measuring the time taken by the client to generate touch messages to be sent to the server for these actions. We obtain this time by collecting network packet traces at the client, filtering them for all touch/mouse message packets with the server as the destination and measuring the time difference between the first and last touch/mouse packet. We also compare *Peek* with VNC by installing an unmodified VNC client and server on LG Nexus 5 smartphone and Samsung Galaxy tablet, respectively. Here, we use a tablet instead of a smartphone as the unmodified server application is incompatible with smartphones. Note that the method to measure the time taken for an action at the client through network level traces eliminates for any bias related to the network conditions and server processing power. We also evaluate *Peek* only on tasks that can be performed on a tablet and a smartphone in the same way, to avoid any bias related to screen size. Therefore, we believe that the server device configuration has no bearing on the action times. Also, to discount for any user bias, we consider the average of 10 measurements for each action.

To benchmark *Peek*, we first consider a primitive action set: tap, double tap, swipe, long press, and drag. We can observe from Fig. 6.7a that *Peek* reduces action times significantly for certain actions. For swipe, long press and drag, the reduction is 80.2%, 39.9%, and 41.8%, respectively. According to [155], an action time increase $>$ 150ms results in

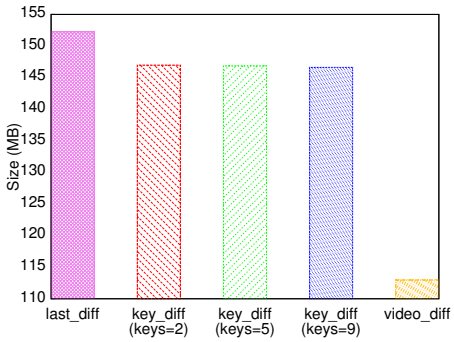
noticeable reduction usability. For tap and double tap, action times are higher for *Peek* by 3ms and 74ms, respectively. This is because unlike the other actions, these actions are mapped as-is, even without multi-touch support. However, this difference does not affect the usability. To evaluate the benefits of *Peek* during regular smartphone usage, we also consider a set of complex actions that span common touch screen usage patterns (Fig. 6.4). For these actions, *Peek* reduces the action time by 62.8% on average (Fig. 6.7b). *Peek* achieves this by eliminating mouse mapping and directly capturing and executing touch actions. We also measured the CPU and memory usage of *Peek* and VNC on the client and observed that *Peek* does not involve any additional overheads. For a proof of concept for context shipping, we also implement proximity sensor context association and verify its function.

Next, we demonstrate the potential of multi-modal compression of *Peek* to reduce the bytes sent over the network for the following applications: (a) Ebay, (b) Google play, (c) BBC, (d) Gmail, (e) Candy Crush, (f) Enterprise Sharepoint. This set is a representative mix that spans some popular application categories. We collect large usage videos for these applications and generate and extract all distinct video frames. We then create synthetic test sessions of size 5500 frames. Each frame is either chosen randomly from the set of distinct images or is the same as the previous image, with equal probability. This dataset represents typical usage behavior wherein a user either sticks with the current view or interacts with it (with 0.5 probability) and hence provides a way to evaluate *Peek* during random user behavior. Since the collection of sizeable real application user traces for many applications is highly intrusive, we use synthetic datasets for evaluation. This is because recording screens and writing them to the storage card, while the user is using an application involves a lot of I/O operations and is a CPU heavy task. We implement and evaluate different modes of multi-modal compression used in *Peek* on this synthetic dataset in Matlab. For *last_diff* and *key_diff* we use Tight PNG (used in VNC), to compress the difference between two frames.

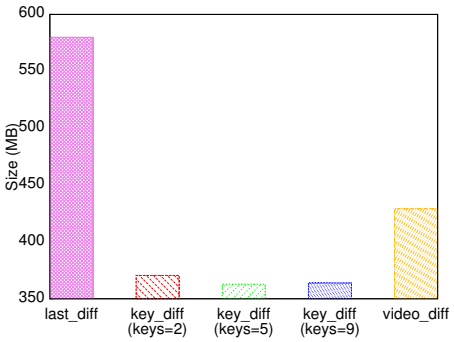


(a) BBC

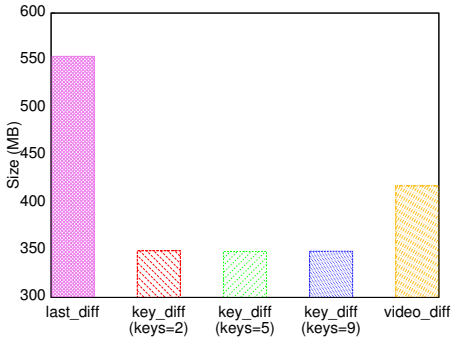
(b) Ebay



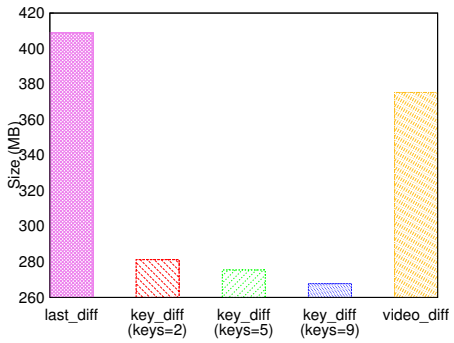
(c) Candy Crush



(d) Gmail



(e) Google Play



(f) Enterprise Sharepoint

Figure 6.8: *Peek* multi-modal compression

Fig. 6.8 shows the post-compression dataset sizes after using different modes of *Peek*'s multi-modal compression for two of the applications. Traditional VNC uses an approach similar to *last_diff*. We observed that using *key_diff* and *video_diff* results in better compression. For example, compared to *last_diff* (used in VNC), *key_diff* with 5 key frames results in a reduction of dataset size reduces compared to *last_diff* (an approach used by traditional VNC) by 27.2%, 36.8%, 37.4%, 31.4% and 32.6% for BBC, Ebay, Gmail, Google play and Enterprise Sharepoint, respectively. For all applications, *last_diff* results in highest post-compression sizes. For Candy Crush, *video_diff* performs the best. This is because it is a game having rapidly changing screen content, with little repetition from the past usage as the user advances to new levels. We can also observe that, increasing the number of stored key frames results in better compression. However, considerable benefits can be achieved by using just two key frames.

CHAPTER 7

INTEGRATED OPERATIONS

In the previous chapters, we discussed the user-aware optimization of three enterprise mobility aspects - (i) Workflow execution, (ii) Content creation and (iii) Collaboration. In this chapter, we discuss how these individual solutions can be integrated within conventional enterprise mobility architectures.

Several user-aware optimizations, including the three contributions of this dissertation, can be integrated with the architecture shown in Figure 1.2 through three enhancements: a Mobility client, a Mobility server and a User-aware datastore. Figure 7.2 shows different components of the Mobility client and server enhancements to integrate *Dejavu*, *Taskr*, *Trackr*, and *Peek* within the enhanced enterprise mobility architecture shown in Figure 7.1. The Mobility client resides on the user's smartphone and the Mobility server is hosted within the enterprise network.

7.0.1 At the enterprise

The Mobility server consists of three major components - the data collector, the data curator, and the orchestrator. The data collector polls the application servers (both on-premise and cloud-based) to collect user's data. For *Dejavu*, the collector obtains user's emails from the mail servers. For *Taskr* and *Trackr*, the collector obtains a user's log of actions from applications like Salesforce, Sharepoint, Peoplesoft, and so on. For *Peek*'s multi-modal frame compression, the collector obtains screen frames from the remote computing sessions. The data curator parses the data obtained from the collector and stores it within the information datastore. For example, for *Dejavu*, the curator processes emails, indexes them and stores them. For *Peek*, the curator computes keyframes periodically and stores them in the datastores. The orchestrator is responsible for managing all processing activities within the

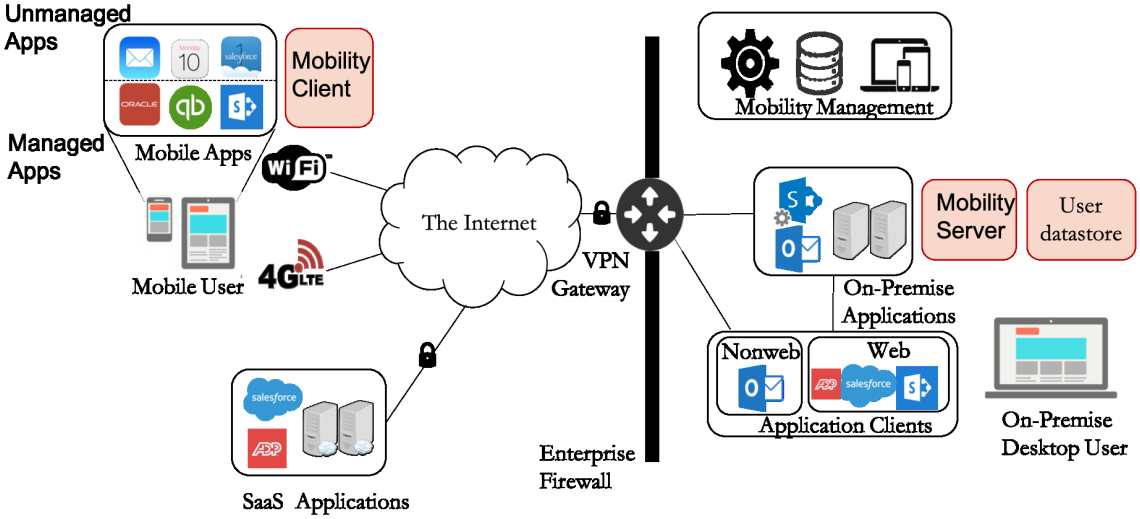


Figure 7.1: Enterprise Mobility Architecture

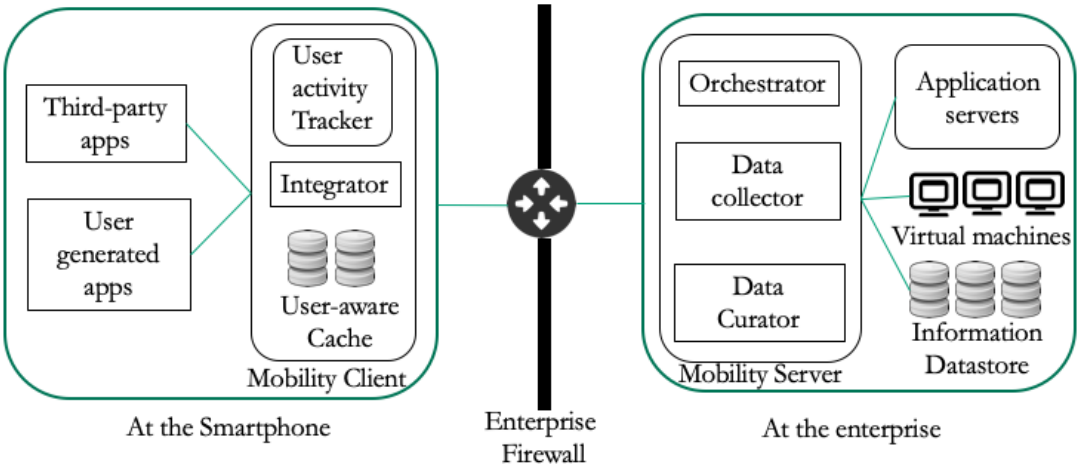


Figure 7.2: Integrated Architecture

mobility server. For *Dejavu*, it calls the suggestions generator to compute and store suggestions for inbox emails within the datastore. For *Taskr*, the orchestrator manages virtual machines with browsers for refactoring based remote computing sessions, transforms the application's UI for mobilization, applies user actions on the virtual machines and communicates with the mobility client when needed. In *Trackr*, the orchestrator manages the persistent storage and updates of fingerprints. In *Peek*, the orchestrator is responsible for maintaining the remote computing session between two clients or a client and a virtual machine. All meta-information related to the users is stored within the Information datastore.

7.0.2 At the smartphone

The mobility client also consists of three major components - A user activity tracker, an integrator and a user-aware cache. The activity tracker module tracks user's actions on apps (i.e., touchscreen gestures and browsing actions) and reports them to the data collector at the mobility server. In addition to user actions, the activity tracker also profiles the device for resource utilization to regulate the computing/network and storage overheads of *Dejavu*, *Taskr*, and *Peek* mobility enhancements. The user-aware cache stores data for mobility enhancements at the device opportunistically. For *Dejavu*, the cache contains suggestions for the most recent emails. For *Taskr*, the cache contains meta-information of the mobilized workflows to reduce communication overheads with the mobility server. In *Peek*, the cache stores the keyframes necessary for multi-modal frame compression. Finally, the integrator is responsible for interacting with existing apps and applying user-aware mobility enhancements to them, i.e. integrating suggestions within email clients for *Dejavu*, compressing remote computing graphical session data for *Peek*, orchestrating the application refactoring based remote computing session for *Taskr* and fingerprinting for *Trackr*.

CHAPTER 8

FUTURE WORK

8.1 Automated reply suggestions

As a part of this dissertation, we developed *Dejavu*, a system for automated email responses to ease the burden of typing these responses on smartphones. As we undertook this research, we identified certain future research directions.

- *Integration with Smart-Reply systems:* A related class of response suggestion systems to *Dejavu* are Smart-Reply systems [69, 71]. They use long-short-term-memory (LSTM) networks to predict the response given the words present in an inbox email. Smart-Reply systems are typically trained on a large amount of email data and are not specific to a particular user. Therefore, the responses suggested by Smart-Reply systems are generic and non-informational. On the other hand, *Dejavu* uses the keywords present in an Inbox email to retrieve suitable suggestions from an Information Database. *Dejavu* further optimizes this retrieval using a variety of heuristics and optimizations. Therefore, *Dejavu*'s user-specific retrieval system is complementary to Smart-Reply systems. Integrating *Dejavu*'s informational suggestions with Smart-Reply's non-informational suggestions would allow a user to select a suitable response from a larger pool of choices.

- *Knowledge Channels: Dejavu* looks for suggestions to a *reply* from information that is present only in an email inbox. However, a knowledge worker encounters several different knowledge channels on a daily basis. These channels could be categorized as read/write (Email), read-only (Dropbox) or write-only (Slack) channels. By adding more information channels to the Information Database of *Dejavu*, the suggestions for replies could be improved. For example, documents from the user's

Dropbox can be fetched using Dropbox APIs [156] by the *Information-Curator*. Different topics in these documents can be added as entries into the *Information Database*. Extending *Dejavu* to other knowledge channels in the future remains an open issue.

- *Evaluation:* We only evaluate *Dejavu* offline by computing *HitRate* for various parameters. However, the usefulness of these suggestions can be truly judged by real users using it daily on their smartphones. The usefulness of suggestions can be captured through opinion score metrics, wherein the users rate every suggestion using a score of 1(not helpful) to 5(very helpful). Evaluating *Dejavu* by distributing a production version of the prototype to a broad set of volunteers and capturing subjective metrics remains an open research issue.

- *Search expansion:* In English, a word can have several synonyms and can be present in different forms. *Dejavu* only deals with the latter by stemming the word and extracting its root. The former problem could be solved by expanding the index used for matching to include all possible synonyms for the words encountered in that index (obtained from a resource such as WordNet [157]). This way, suggestions containing words that have different roots but similar meaning can be retrieved as matches.

8.2 Do-it-yourself application mobilization

- *Security:* Most enterprise applications require the user to log in (either explicitly or through a single sign-on service) before any workflow can be executed. The requirement of log in usually does not restrict the number of workflows that qualify as spot tasks as the username and password can be treated as fixed parameters. The login username and password are required by *Taskr* to execute workflows on most enterprise applications. These parameters constitute sensitive information and need to be handled carefully. The login parameters constitute sensitive information and

can be encrypted and stored on the local device using services like keychain API for iOS. When the spot task has to be executed, these parameters can be encrypted and sent to the server using transport security such as SSL. Alternately, this sensitive data can be stored in the cloud isolated within the enterprise network and hence be protected by enterprise firewalls. The user can then be restricted to using *Taskr* within the enterprise network. If the application server allows it, a continuous login session can be maintained at the *Taskr-server* using the stored username and password.

- *Evaluation:* *Taskr* requires accurate fingerprinting of UI elements to execute the workflow. While we discuss the fingerprint technique used by *Taskr* in Section 4.2 and implement it in the prototype, we do not evaluate it for correctness. However, we observe that for the different spot tasks considered in Section 4.3.2, the fingerprinting is accurate. We plan to investigate this in the future. We implemented *Taskr-client* and server for twitter, email and native app usage modalities. However, we only conduct subjective tests on the native mobile app modality. We plan to implement a few other modalities and extend the testing in the future.

- *Extraction rules and Translation tables:* *Taskr* relies on manually constructed rules for information extraction and fixed translation tables. For the prototype, we constructed these rules for most elements defined by the HTML5 standard. However, many web applications use elements defined by third party UI frameworks. We plan to extend these rules for some popular UI frameworks used by web applications. *Taskr* performs one to one translation between web UI and smartphone native UI. However, some platforms allow the creation of macros to bundle several UI element interactions into one interaction. Extension of *Taskr* to consider many to one or one to many translations in the context of these macros is an open issue.

- *Extension to other workflows:* *Taskr* helps users mobilize simple workflows that can be described as spot tasks. This restriction limits the number of workflows that can

be mobilized. We plan to relax these restrictions to include workflows that can be described as a sequence of spot tasks, and also other general workflows in the future.

8.3 Robust front-end APIfication

- *Software design choice:* In this paper, we designed *Trackr* to be a browser extension. However, the principles of *Trackr* are not restricted to this design choice. Alternatively, *Trackr* can also be implemented as a javascript library that the web applications can include to avail fingerprinting services;

- *Reactive vs. Proactive updates:* *Trackr* updates the stored fingerprints reactively upon every subsequent visit to the web application by the user. While this approach could work well if the pages are frequently visited by the user, a reactive approach wherein *Trackr* periodically updates the fingerprint is more suited for infrequently accessed pages;

- *Identification of the web page:* *Trackr* stores the fingerprints in a database indexed by the name of the tracker and a URL of the web page. However, it is possible for some web pages to have a dynamic URL ,e.g. news articles. In this case, a better indexing mechanism would be to create a fingerprint for the page itself, independent of the URL. One way to achieve this is to select a subset of elements whose presence definitively identifies the web page. We plan to address these issues in the future;

- *Complexity:* To find an element, *Trackr* uses DOM access methods to retrieve all elements of the same type and compares their fingerprints. The worst case complexity is proportional to the size of the DOM. However, our observations from experiments indicate that the retrieval does not add any noticeable delays. We plan on performing a more formal study on the complexity in the future.

8.4 Mobile-to-Mobile Remote computing for smartphones

- *Remote computing between heterogeneous devices Peek* assumes that both the server and client in a mobile-to-mobile remote computing session are homogeneous devices with same screen aspect ratios. While this is a reasonable assumption within enterprises adopting a corporate owned personally enabled device (COPE) policy, it fails with a bring your own device (BYOD) policy, where the employees own a variety of devices. If the client and server have different aspect ratios, the touch actions at the client have to be scaled to the aspect ratio of the server before they can be applied at the server. Modeling and estimation of the scaling factor and implementation within *Peek* is an open research direction.

- *Network delays Peek* assumes the remote computing messages are transmitted between the client and the server over a reliable transport layer protocol like TCP, to ensure the receipt of packets in order. However, when the underlying network is subject to bursty losses, the network delays can be very high resulting in an unresponsive session. On the other hand, using transport layer protocols like UDP over unreliable networks may result in loss of information due to packet losses. Investigating the impact of network protocols and conditions in the presence of unreliable networks is a potential research direction.

CHAPTER 9

CONCLUSIONS

Application mobilization, or delivering an enterprise employee the ability to rely on their mobile devices to continue to perform their business functions even when away from the Desktop, is seen as a game changer to boost productivity among enterprise employees. However, the potential benefits of enterprise mobility are yet to be realized. A vast majority of enterprise applications are either not mobilized, or are unusable for enterprise employees. We argue that one of the major factors contributing to the poor adoption of smartphone apps within the enterprise is the process of *defeaturization*, wherein a subset of features within complex Desktop applications are ported into a smartphone app. Defeaturization in enterprises has been done in a user-unaware fashion with the enterprises or the software vendors choosing which subset of features to include in a smartphone app. This results in users facing several issues.

In this thesis, we focused on two of the issues that hinder the true adoption of enterprise mobility - the heavy task burden of accomplishing tasks, and the unavailability of critical job functions. We argued that user-aware defeaturization can mitigate these issues. In this context, we explored four different research directions.

In Chapter 3 we considered the problem of automated information suggestions to assist in *reply* construction for Email on mobile devices. The basic premise of the work is that a significant portion of the information content of a *reply* is likely to be present in prior emails. Through an analysis of multiple public Email datasets, we first established that there is considerable redundancy between *replies* and previous emails. We then presented a simple user-aware solution called *Dejavu* that uses keyword matching to provide automated suggestions during *reply* construction, using information present in the user's mailbox. We further proposed *Dejavu* ++, an optimized version of *Dejavu* to reduce the complexity of

finding suggestions and improve the relevancy of the suggested replies. When applied to the same datasets, we showed that *Dejavu* and *Dejavu* ++ have the potential to reduce the heavy task burden of typing emails on a smartphone keyboard.

In Chapter 4, we considered the problem of mobilizing Spot Tasks, a particular category of workflows within web-based enterprise applications that can be finished by interacting with only one page of the application. We presented Taskr, a do-it-yourself mobilization solution that users, regardless of their skills, can rely on to mobilize their spot tasks robustly. Taskr uses remote computing with application refactoring to achieve code-less mobilization and allows for flexible mobile delivery wherein users can execute their spot tasks through Twitter, Email or a native mobile app. We implemented a prototype of Taskr and show through user studies that it has the potential to reduce task burden significantly.

In Chapter 5, we proposed *Trackr*, an algorithm to reliably track UI elements within a web application for robust API creation. Accurate tracking of elements within a web application is crucial for refactoring based application mobilization services. We introduced the principle of quorum fingerprinting used by *Trackr* to create unique identities for the tracked elements and presented optimizations designed to increase its robustness. We evaluated *Trackr* over four popular web applications to show attractive benefits. Finally, we discussed *Trackr*'s application through three uses cases.

In Chapter 6, we presented a brief overview of *Peek*, a smartphone to smartphone remote computing protocol with multi-touch support, context association, and a user-aware multi-modal frame compression. *Peek* allows users to collaborate even in the absence of native multi-user support. We evaluated Peek and show that it reduces the time taken to perform specific actions by over 60% and also reduces the number of bytes transmitted into the network by over 30%, compared to traditional VNC.

Finally, in Chapter 7, we discussed how the four complimentary user-aware solutions proposed in the earlier chapters could work in tandem in the context of a simplified enterprise mobility architecture.

REFERENCES

[1] M. Turk, *Employees say smartphones boost productivity by 34 percent: Frost and sullivan research*, https://insights.samsung.com/2016/08/03/employees-say-smartphones-boost-productivity-by-34-percent-frost-sullivan-research/.

[2] *Bring your own device: What you need to know*, https://info.sapho.com/hubfs/Resources/BYOD_Checklist.pdf.

[3] J. Bradley, *New analysis: Comprehensive byod implementation increases productivity, decreases costs*, https://blogs.cisco.com/news/new-analysis-comprehensive-byod-implementation-increases-productivity-decreases-costs.

[4] S. Solomon, *Byod and enterprise mobility market to exceed $284 billion by 2019*, http://blogs.air-watch.com/2014/04/marketsandmarkets-expects-byod-enterprise-mobility-market-grow-284-7-billion-2019/.

[5] *Salesforce*, https://www.salesforce.com/.

[6] *Aaptivo*, https://www.apptivo.com/.

[7] *Hubspot*, https://www.hubspot.com/.

[8] *Zoho*, https://www.zoho.com/.

[9] *Adp*, https://www.adp.com/.

[10] *Kronos*, https://www.kronos.com/.

[11] *Zenefits*, https://www.zenefits.com/.

[12] *Oracle peoplesoft*, https://www.oracle.com/applications/peoplesoft/.

[13] *Oracle netsuite*, http://www.netsuite.com/portal/home.shtml.

[14] *Workday*, https://www.workday.com/en-us/homepage.html.

[15] *Sap erp*, https://www.sap.com/products/enterprise-management-erp.html.

[16] *Microsoft dynamics*, https://dynamics.microsoft.com/en-us/.

[17] *Tableau*, https://www.tableau.com/.

[18] *Domo*, https://www.domo.com/.

[19] *Oracle business intelligence*, https://www.oracle.com/solutions/business-analytics/business-intelligence/.

[20] *Sap crystal reports*, https://www.sap.com/products/crystal-reports.html.

[21] *Adobe experience manager*, https://www.adobe.com/marketing/experience-manager.html.

[22] *Microsoft sharepoint*, https://products.office.com/en-us/sharepoint/collaboration.

[23] *Sitecore*, https://www.sitecore.com/.

[24] *Avaya*, https://www.avaya.com/en/.

[25] *Cisco unified communications manager*, https://www.cisco.com/c/en/us/products/unified-communications/unified-communications-manager-callmanager/index.html.

[26] *Skype for business*, https://www.skype.com/en/business/.

[27] *Quickbooks*, https://quickbooks.intuit.com/.

[28] *Sap business one*, https://www.sap.com/products/business-one.html.

[29] *Ibm maximo*, https://www.ibm.com/products/maximo.

[30] *Abb ability asset suite*, https://new.abb.com/enterprise-software/asset-optimization-management/asset-suite-eam.

[31] *Mvp plant*, http://mvpplant.com/.

[32] *Supply chain planning*, https://www.sap.com/products/digital-supply-chain/scm.html.

[33] *Oracle supply chain management*, https://www.sap.com/products/digital-supply-chain/scm.html.

[34] *Jda*, https://jda.com/.

[35] *Atlassian*, https://www.atlassian.com/.

[36] *Collabnet versionone*, https://www.collab.net/.

[37] *Information technology (it) spending on enterprise software worldwide, from 2009 to 2019 (in billion u.s. dollars)*, https://www.statista.com/statistics/203428/total-enterprise-software-revenue-forecast/.

[38] *Smartphone penetration rate as share of the population in the united states from 2010 to 2022*, https://www.statista.com/statistics/201183/forecast-of-smartphone-penetration-in-the-us/.

[39] S. Zhao, J. Ramos, J. Tao, Z. Jiang, S. Li, Z. Wu, G. Pan, and A. K. Dey, "Discovering different kinds of smartphone users through their application usage behaviors," in *Proceedings of the 2016 ACM International Joint Conference on Pervasive and Ubiquitous Computing*, ser. UbiComp '16, Heidelberg, Germany: ACM, 2016, pp. 498–509, ISBN: 978-1-4503-4461-6.

[40] *Worldwide netskope cloud report - january 2017*, https://resources.netskope.com/cloud-reports/january-2017-worldwide-cloud-report.

[41] *Enterprise app trends report*, https://arxan.com/enterprise-app-trends.

[42] *The state of enterprise mobility*, http://synchronoss.com/wp-content/uploads/The_State_of_Enterprise_Mobility_Whitepaper.pdf.

[43] D. Roberts, *Usability and mobile devices*, https://www.usability.gov/get-involved/blog/2010/05/mobile-device-usability.html.

[44] A. Neagu, *Figuring the costs of custom mobile app development*, https://www.formotus.com/blog/figuring-the-costs-of-custom-mobile-business-app-development.

[45] *The social economy: Unlocking value and productivity through social technologies*, http://www.mckinsey.com/industries/high-tech/our-insights/the-social-economy.

[46] S. Whittaker and C. Sidner, "Email overload: Exploring personal information management of email," in *Proceedings of the SIGCHI conference on Human factors in computing systems*, ACM, 1996, pp. 276–283.

[47] R. Bergman, M. Griss, and C. Staelin, "A personal email assistant," 2002.

[48] K. Mock, "An experimental framework for email categorization and management," in *Proceedings of the 24th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2001, pp. 392–393.

[49] G. Carenini, R. T. Ng, and X. Zhou, "Summarizing emails with conversational cohesion and subjectivity.," in *ACL*, vol. 8, 2008, pp. 353–361.

[50] S. Joty, G. Carenini, G. Murray, and R. T. Ng, "Exploiting conversation structure in unsupervised topic segmentation for emails," in *Proceedings of the 2010 Conference on Empirical Methods in Natural Language Processing*, Association for Computational Linguistics, 2010, pp. 388–398.

[51] M. Dredze, H. M. Wallach, D. Puller, and F. Pereira, "Generating summary keywords for emails using topics," in *Proceedings of the 13th international conference on Intelligent user interfaces*, ACM, 2008, pp. 199–206.

[52] W. W. Cohen, V. R. Carvalho, and T. M. Mitchell, "Learning to classify email into"speech acts".," in *EMNLP*, 2004, pp. 309–316.

[53] V. R. Carvalho and W. W. Cohen, "On the collective classification of email speech acts," in *Proceedings of the 28th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2005, pp. 345–352.

[54] S. Joty, G. Carenini, and C.-Y. Lin, "Unsupervised modeling of dialog acts in asynchronous conversations," in *IJCAI Proceedings-International Joint Conference on Artificial Intelligence*, vol. 22, 2011, p. 1807.

[55] D. Aberdeen, O. Pacovsky, and A. Slater, "The learning behind gmail priority inbox," in *LCCC: NIPS 2010 Workshop on Learning on Cores, Clusters and Clouds*, 2010.

[56] *Priority inbox*, `https://support.google.com/mail/answer/186531?hl=en`.

[57] S. Scerri, B. Davis, S. Handschuh, and M. Hauswirth, "Semanta–semantic email made easy," in *The Semantic Web: Research and Applications*, Springer, 2009, pp. 36–50.

[58] K. Balog, L. Azzopardi, and M. De Rijke, "Formal models for expert finding in enterprise corpora," in *Proceedings of the 29th annual international ACM SIGIR conference on Research and development in information retrieval*, ACM, 2006, pp. 43–50.

[59] D. Zhang, "Web content adaptation for mobile handheld devices," *Commun. ACM*, vol. 50, no. 2, pp. 75–79, Feb. 2007.

[60] D. Jurafsky and J. H. Martin, *Speech and Language Processing: An Introduction to Natural Language Processing, Computational Linguistics, and Speech Recognition*, 1st. Upper Saddle River, NJ, USA: Prentice Hall PTR, 2000, ISBN: 0130950696.

[61] D. Chen, A. Fisch, J. Weston, and A. Bordes, "Reading wikipedia to answer open-domain questions," *CoRR*, vol. abs/1704.00051, 2017. arXiv: `1704.00051`.

[62] *Watson discovery*, https://www.ibm.com/watson/services/discovery/.

[63] J. Devlin, M. Chang, K. Lee, and K. Toutanova, "BERT: pre-training of deep bidirectional transformers for language understanding," *CoRR*, vol. abs/1810.04805, 2018. arXiv: 1810.04805.

[64] A. W. Yu, D. Dohan, M. Luong, R. Zhao, K. Chen, M. Norouzi, and Q. V. Le, "Qanet: Combining local convolution with global self-attention for reading comprehension," *CoRR*, vol. abs/1804.09541, 2018. arXiv: 1804.09541.

[65] Y. Yu, W. Zhang, K. S. Hasan, M. Yu, B. Xiang, and B. Zhou, "End-to-end reading comprehension with dynamic answer chunk ranking," *CoRR*, vol. abs/1610.09996, 2016. arXiv: 1610.09996.

[66] M. Hu, F. Wei, Y. Peng, Z. Huang, N. Yang, and M. Zhou, "Read + verify: Machine reading comprehension with unanswerable questions," *CoRR*, vol. abs/1808.05759, 2018. arXiv: 1808.05759.

[67] A. Fader, L. Zettlemoyer, and O. Etzioni, "Paraphrase-driven learning for open question answering," in *Proceedings of the 51st Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Sofia, Bulgaria: Association for Computational Linguistics, 2013, pp. 1608–1618.

[68] J. Berant and P. Liang, "Semantic parsing via paraphrasing," in *Proceedings of the 52nd Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, Baltimore, Maryland: Association for Computational Linguistics, 2014, pp. 1415–1425.

[69] *Computer, respond to this email*, http://googleresearch.blogspot.com/2015/11/computer-respond-to-this-email.html.

[70] *Ios quicktype*, http://www.apple.com/my/ios/whats-new/quicktype/.

[71] *Use intelligent technology in outlook.com*, https://support.office.com/en-us/article/use-intelligent-technology-in-outlook-com-edf8204f-2bb9-45e1-8620-fc43a2ecdba3.

[72] *The enterprise platform for digital transformation*, http://www.anypresence.com/.

[73] *Appcelerator*, http://www.appcelerator.com/.

[74] N. Bila, T. Ronda, I. Mohomed, K. N. Truong, and E. de Lara, "Pagetailor: Reusable end-user customization for the mobile web," in *Proceedings of the 5th International Conference on Mobile Systems, Applications and Services*, ser. MobiSys '07, San Juan, Puerto Rico, 2007, ISBN: 978-1-59593-614-1.

[75] H. Li, P. Li, S. Guo, X. Liao, and H. Jin, "Modeap: Moving desktop application to mobile cloud service," *Mobile Networks and Applications*, vol. 19, no. 4, pp. 563–571, 2014.

[76] I. Mohomed, "Enabling mobile application mashups with merlion," in *Proceedings of the Eleventh Workshop on Mobile Computing Systems & Applications*, ser. HotMobile '10, Annapolis, Maryland, 2010.

[77] *Feed circuit*, http://feedcircuit.garage.maemo.org/.

[78] J. Nichols and T. Lau, "Mobilization by demonstration: Using traces to re-author existing web sites," in *Proceedings of the 13th International Conference on Intelligent User Interfaces*, ser. IUI '08, Gran Canaria, Spain: ACM, 2008, pp. 149–158, ISBN: 978-1-59593-987-6.

[79] A. Moshchuk, S. D. Gribble, and H. M. Levy, "Flashproxy: Transparently enabling rich web content via remote execution," in *Proceedings of the 6th International Conference on Mobile Systems, Applications, and Services*, Breckenridge, CO, USA, 2008, pp. 81–93.

[80] N. B. Niraula, A. Stent, H. Jung, G. D. Fabbrizio, I. D. Melamed, and V. Rus, "Forms2dialog: Automatic dialog generation for web tasks," in *2014 IEEE Spoken Language Technology Workshop (SLT)*, Dec. 2014, pp. 608–613.

[81] M. Nebeling, M. Speicher, and M. Norrie, "W3touch: Metrics-based web page adaptation for touch," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '13, Paris, France: ACM, 2013, pp. 2311–2320, ISBN: 978-1-4503-1899-0.

[82] Y. Potla, R. Annadi, J. Kong, G. Walia, and K. Nygard, "Adapting web page tables on mobile devices," *Int. J. Handheld Comput. Res.*, vol. 3, no. 1, pp. 1–22, Jan. 2012.

[83] C.-H. Yu and R. C. Miller, "Enhancing mobile browsing and reading," in *CHI '11 Extended Abstracts on Human Factors in Computing Systems*, ser. CHI EA '11, Vancouver, BC, Canada: ACM, 2011, pp. 1783–1788, ISBN: 978-1-4503-0268-5.

[84] S. Wang, W. Dou, G. Wu, J. Wang, C. Gao, J. Wei, and T. Huang, "Towards web application mobilization via efficient web control extraction," in *Proceedings of the 7th Asia-Pacific Symposium on Internetware*, ser. Internetware '15, Wuhan, China: ACM, 2015, pp. 21–29, ISBN: 978-1-4503-3641-3.

[85] Y. Ma, Y. Fang, X. Zhu, X. Liu, and G. Huang, "Mobitran: Tool support for refactoring pc websites to smart phones," in *Proceedings Demo & Poster Track of ACM/IFIP/USENIX International Middleware Conference*, ser. MiddlewareDPT '13, Beijing, China: ACM, 2013, 6:1–6:2, ISBN: 978-1-4503-2549-3.

[86] W. W. W. C. (W3C), *Xml path language (xpath) version 3.1*, `https://www.w3.org/TR/xpath-31/`, 2017.

[87] G. Gottlob and et.al., "Efficient algorithms for processing xpath queries," *ACM Trans. Database Syst.*, vol. 30, no. 2, pp. 444–491, '05.

[88] C.-Y. Chan, P. Felber, M. Garofalakis, and R. Rastogi, "Efficient filtering of xml documents with xpath expressions," *The VLDB Journal*, vol. 11, no. 4, pp. 354–379, Dec. 2002.

[89] L. Yi and B. Liu, "Web page cleaning for web mining through feature weighting," in *Proceedings of the 18th International Joint Conference on Artificial Intelligence*, ser. IJCAI'03, Acapulco, Mexico, 2003, pp. 43–48.

[90] S. Sanadhya, "Ultra-mobile computing: Adapting network protocol and algorithms for smartphones and tablets," PhD thesis, Georgia Institute of Technology, 2013.

[91] L. Zhang, M. Li, N. Dong, and Y. Wang, "An improved dom-based algorithm for web information extraction," *JOURNAL OF INFORMATION &COMPUTATIONAL SCIENCE*, vol. 8, no. 7, pp. 1113–1121, 2011.

[92] J. P. Cohen, W. Ding, and A. Bagherjeiran, "Semi-supervised web wrapper repair via recursive tree matching," *CoRR*, 2015.

[93] S. Zheng, R. Song, and J.-R. Wen, "Template-independent news extraction based on visual consistency," in *AAAI*, vol. 7, 2007, pp. 1507–1513.

[94] D. Cai, S. Yu, J.-R. Wen, and W.-Y. Ma, "Vips: A vision-based page segmentation algorithm," Tech. Rep., Nov. 2003, p. 28.

[95] *Remote Desktop Protocol*, `http://msdn.microsoft.com/en-us/library/aa383015(v=vs.85).aspx`.

[96] *The rfb protocol*, `http://www.realvnc.com/docs/rfbproto.pdf`.

[97] *Citrix receiver*, `https://www.citrix.com/products/receiver/`.

[98] C.-L. Tsao, S. Kakumanu, and R. Sivakumar, "Smartvnc: An effective remote computing solution for smartphones," in *Proceedings of the 17th Annual International Conference on Mobile Computing and Networking*, Las Vegas, Nevada, USA, 2011, pp. 13–24, ISBN: 978-1-4503-0492-4.

[99] R. A. Baratto, S. Potter, G. Su, and J. Nieh, "Mobidesk: Mobile virtual desktop computing," in *Proceedings of the 10th Annual International Conference on Mobile Computing and Networking*, ser. MobiCom '04, Philadelphia, PA, USA: ACM, 2004, pp. 1–15.

[100] *Gaming anywhere*, http://gaminganywhere.org/index.html.

[101] B. Joveski *et al.*, "Semantic multimedia remote display for mobile thin clients," *Multimedia Systems*, vol. 19, no. 5, pp. 455–474, 2013.

[102] F. Liu, J. Wang, and H. Bai, "On the compression of hyperspectral data," *IT CoNvergence PRActice (INPRA)*, vol. 1, no. 4, pp. 39–49, Dec. 2013.

[103] S. Radicati, "E-mail statistics report, 2014–2018," *Paolo Alto: The Radicati Group Inc*, 2014.

[104] F. Kooti, L. M. Aiello, M. Grbovic, K. Lerman, and A. Mantrach, "Evolution of conversations in the age of email overload," in *Proceedings of the 24th International Conference on World Wide Web*, ser. WWW '15, Florence, Italy: ACM, 2015, pp. 603–613, ISBN: 978-1-4503-3469-3.

[105] *The enron email dataset*, https://www.cs.cmu.edu/~./enron/.

[106] D. Oard, W. Webber, D. Kirsch, and S. Golitsynskiy, *Avocado research email collection*, https://catalog.ldc.upenn.edu/LDC2015T03.

[107] *Enron employee roles*, http://www.inf.ed.ac.uk/teaching/courses/tts/assessed/roles.txt.

[108] *Talon*, https://github.com/mailgun/talon.

[109] *Natural language toolkit*, www.nltk.org.

[110] D. M. Blei, A. Y. Ng, and M. I. Jordan, "Latent dirichlet allocation," *J. Mach. Learn. Res.*, vol. 3, pp. 993–1022, Mar. 2003.

[111] C. H. Papadimitriou, H. Tamaki, P. Raghavan, and S. Vempala, "Latent semantic indexing: A probabilistic analysis," in *Proceedings of the Seventeenth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, ser. PODS '98, Seattle, Washington, USA: ACM, 1998, pp. 159–168, ISBN: 0-89791-996-3.

[112] T. Hofmann, "Probabilistic latent semantic indexing," in *Proceedings of the 22Nd Annual International ACM SIGIR Conference on Research and Development in Information Retrieval*, ser. SIGIR '99, Berkeley, California, USA: ACM, 1999, pp. 50–57, ISBN: 1-58113-096-1.

[113] S. Geman and D. Geman, "Stochastic relaxation, gibbs distributions, and the bayesian restoration of images," *IEEE Transactions on Pattern Analysis and Machine Intelligence*, vol. PAMI-6, no. 6, pp. 721–741, 1984.

[114] *Amazon ec2 - virtual server hosting*, https://aws.amazon.com/?nc2=h_lg.

[115] *K-9 mail*, https://play.google.com/store/apps/details?id=com.fsck.k9&hl=en.

[116] *Swype for android*, https://www.nuance.com/mobile/mobile-applications/swype/android.html.

[117] *Scikit-learn*, https://scikit-learn.org/stable/.

[118] *Employees say smartphones boost productivity by 34 percent*, https://goo.gl/PmEUys.

[119] *Gartner survey shows that mobile device adoption in the workplace is not yet mature*, https://www.gartner.com/newsroom/id/3528217.

[120] *Mobile workforce to drive further enterprise change in 2017*, https://goo.gl/uWHqGm.

[121] *2017 trends in enterprise mobility*, https://goo.gl/3M2Ruv.

[122] *Supportcentral*, https://supportcentral.ge.com/siteminderagent/forms/sso_login.asp.

[123] *Kinvey*, https://www.kinvey.com/.

[124] *Google trends on web platforms*, https://goo.gl/qqy558.

[125] T. Richardson and J. Levine, "The remote framebuffer protocol," 2011.

[126] *Remote desktop protocol*, http://msdn.microsoft.com/en-us/library/aa383015(VS.85).aspx.

[127] *Ext js*, https://www.sencha.com/products/extjs.

[128] *Sakai*, https://sakaiproject.org/.

[129] *Bootstrap. the world's most popular mobile first and responsive front-end framework*, http://getbootstrap.com/.

[130] *Xui, a tiny javascript framework for mobile web apps*, https://github.com/xui/xui.

[131] *Dojo toolkit*, https://dojotoolkit.org/.

[132] *Express - node.js web framework*, https://expressjs.com/.

[133] *New relic*, https://newrelic.com/.

[134] *Backbone.js*, http://backbonejs.org/.

[135] *Require.js*, http://requirejs.org/.

[136] *Underscore.js*, http://underscorejs.org/.

[137] *D3.js*, https://d3js.org/.

[138] *Modernizr.js*, https://modernizr.com/.

[139] *Moment.js*, https://momentjs.com/.

[140] *Nvd3.js*, http://nvd3.org/.

[141] *Fancybox*, http://fancybox.net/.

[142] *Google maps apis*, https://developers.google.com/maps/showcase/,

[143] *Javascript html dom*, https://www.w3schools.com/js/js_htmldom.asp.

[144] E. Adar, J. Teevan, S. T. Dumais, and J. L. Elsas, "The web changes everything: Understanding the dynamics of web content," in *Proceedings of the Second ACM International Conference on Web Search and Data Mining*, ser. WSDM '09, Barcelona, Spain, 2009, pp. 282–291, ISBN: 978-1-60558-390-7.

[145] *Selenium*, http://www.seleniumhq.org/.

[146] *Ifttt*, https://ifttt.com/.

[147] *Capriza*, https://www.capriza.com/.

[148] *Peoplesoft applications overview*, http://www.oracle.com/us/products/applications/peoplesoft-enterprise/overview/index.html.

[149] *Sharepoint*, https://products.office.com/en-us/sharepoint/collaboration.

[150] A. Yilmaz, O. Javed, and M. Shah, "Object tracking: A survey," *Acm computing surveys (CSUR)*, vol. 38, no. 4, p. 13, 2006.

[151] *Cisco collaboration work practice study*, https://www.cisco.com/c/dam/en/us/solutions/collaboration/collaboration-sales/cwps_full_report.pdf.

[152] *The rfb protocol*, http://www.realvnc.com/docs/rfbproto.pdf.

[153] *Citrix receiver*, https://www.citrix.com/products/receiver/.

[154] *Pcoip*, http://www.teradici.com/pcoip-technology.

[155] N. Tolia *et al.*, "Quantifying interactive user experience on thin clients," *Computer*, vol. 39, no. 3, pp. 46–52, Mar. 2006.

[156] *Dropbox api*, https://www.dropbox.com/developers.

[157] *Wordnet. a lexical database for english*, https://wordnet.princeton.edu/.