A Model for Contextual Data Sharing in Smartphone Applications

Harshvardhan J. Pandit

A thesis submitted to the National University of Ireland, Cork
in fulfillment of the requirements for the degree of
Master of Science by Research

June 2015

Research Supervisor	Adrian O'Riordan
Head of Department	Prof. Barry O'Sullivan

Department of Computer Science,
National University of Ireland, Cork.

Abstract

The advent of smartphones as a computing device has resulted in a shift in focus towards the design and development of smartphone applications or apps, that allow the user to complete a wide range of tasks on their devices. The users depend on apps installed on their smartphones to access services such as emails, photos, music, browsing, messaging and telephony. However, the overall user experience is disjointed as users are required to use multiple apps to complete a task where each app requires the user to enter the same information as the apps cannot share the data contextually.

This thesis investigates how smartphone apps can perform contextual data sharing with an emphasis on practical integration into the existing platforms and app models. The identification of information and its associated context is necessary to create context definitions that allow different apps to identify the context of the shared data. An approach to model the Context Definitions using computer science concepts such as object-oriented data structures provides flexibility. A context datastore is defined to store and share contextual information between apps, which creates an independence between apps for acquiring information and provides compatibility with the existing security models on various platforms. The model allows apps to retrieve contextual data in a simple and efficient manner without interacting directly with the other apps.

This thesis explains the author's hypothesis of creating contextual services in apps based on the availability of contextual information on a smartphone device. An implementation of the model proving the hypothesis is presented on Android using native tools and technologies available on the platform. The demonstration aims to show the viability of the model through use cases, evaluations and performance metrics.

Finally, the author provides recommendation for developers in adoption of the model, and the efforts required to integrate the implementation into existing platforms and apps. Further research avenues are identified that define the future of research in this area.

Acknowledgments

'If we knew what it was we were doing,
it would not be called research, would it?"
- Albert Einstein

I started this research course full of eagerness and excitement. There came many times when I was stuck, stalled, or even frustrated. The support I received from my parents, friends, and colleagues went a long way in keeping me firmly rooted and determined to complete my work. I would like to acknowledge and thank them for everything they have done in support of me and my work.

I would like to thank my supervisor, Adrian O'Riordan, without whom my work and research would never have been completed. His guidance, support, patience, and helpfulness were of great help.

I would also like to thank my advisor, John O'Mullane, who gave support, time, and valuable insight that motivated me to do better.

I would like to thank my colleagues from the Computer Science department, who provided companionship and gave valuable feedback. I would like in particular to thank Gareth W. Young, the other inmate in Lab 2.08, for helping me when I was stuck, tolerating me, and for providing guidance. I also would like to thank Cathal Hoare, and Michael O'Sullivan for their valuable insights, discussions and guidance. I would also like to thank the other people I met in lab 2.08 - Tracey, Katie, and Tamara for their company.

I would like to thank the people I met and helped as a Demonstrator in the various labs. They taught me patience, perseverance, and debugging skills. I would like to thank Derek Bridge, John Morrison, and Leslie Brookes for giving me this opportunity.

A thank you also for everyone in the Computer Science administration office for all their support and help with a variety of issues.

I would like to thank all of those whom I collaborated with for discussions, articles, and research activities. I would also like to thank anyone that I met in any capacity during my journey.

A big thank you to my friends, who constantly kept me motivated and supported me in whatever I did. Thanks to the UCC Indian Society and the wonderful people I met through it. Thanks to the Cork Creative Writing Group, who helped me relax and indulge in my hobbies. A special thank you to Ankit, Apra, Apurva, Aseem, Bhavana, Brijesh, Chandan, Fawaz, Felix, Gauri, Harshada, Justnya, Kara, Kate, Kevin, Lata, Michael, Mitesh, Monish, Parag, Preeti, Purva, Rohit, Sahil, Supriya, Urvashi, and many others who I haven't named; Thank you for being there for me.

Last, but by no means least, I would like to thank my family, in particular my parents. Without them this work would not have been possible and their everlasting support, encouragement and belief in me goes way beyond the call of duty.

This thesis is dedicated to my wonderful parents,

Nilima and Jitendra Pandit.

Thank you both for everything.

Declaration

No portion of the work referred to in this thesis has been submitted in support of an application for another degree or qualification of this or any other university or other institution of learning.

Signed:
Harshvardhan J. Pandit

Contents

List of Tables

List of Figures

Part I

Introduction

1

Introduction

"Focus on the user and all else will follow."
– Google's 9 Principles of Innovation

Smartphones are the most popular computing device [1] due to their ability to offer features such as web browsing, navigation and media consumption along with communication. The most popular smartphone operating systems in use today are Android and iOS [2], which grew in popularity within a short span of time.[1] The features in a smartphone are exploited by apps, which are third-party native applications that provide functionality and utility on the smartphone. Within a short time, the number of smartphone applications have increased tremendously[2] and has led to the formation of an ecosystem comparable to software on a traditional personal computer. The term *app* is a shortening of the term *application software*, which became popular in 2009 when technology columnist David Pogue said that newer smartphones could be nicknamed "*app phones*" [6], and when app was listed as "Word of the Year" by the American Dialect Society [7] in 2010. In a study done in 2012, *comScore* reported that more users used apps for a service than using its website [8]. The popularity of apps has lead to various studies, with research showing that usage of mobile apps strongly correlates with user context and depends on user's location and time of the day [9]. Thus, smartphone apps have an important relation to the user's context, as they perform a large number of tasks on a device that users keep with them at all times.

Different kinds of applications have the potential to utilize different kinds of contextual information, but are restricted when it comes to sharing this information with the

[1]The first iOS device was the iPhone, released in 2007; and the first Android device was the HTC Dream, released in 2008

[2]The Google Play Store (Android) and the App Store (iOS) have over 1.3 million apps published [3, 4] and over 50 billion downloads [5, 4].

other applications. Smartphone applications use *sandboxing* [10], a security model that prevents an application from accessing or changing another app's data. A consequence of this approach is the restrictions in using an app's data to create contextual services on the device. Since apps cannot easily share information with each other, the user is required to input the same information multiple times in different apps used for a task where various apps handle the different steps related to the task. This restrictive data sharing between the apps limits the availability of contextual information on the smartphone as data generated or entered within an app is not available outside the app. The information locked within the apps can be utilized for generating contextual use cases and capabilities that can help users complete their tasks in a faster and simpler way. The Contextual Data Sharing Model described in this research alleviates this problem by storing related contextual information from different apps, which can be accessed and utilized to develop contextual services. This increases the usability of the apps and prevents duplication of effort and information while making it easier for users to perform the related steps belonging to a task.

1.1 Existing Problems and Limitations

Applications that utilize context generally focus on using a limited set of contextual information. The various previous approaches [11] related to using contextual information on a smartphone have focused on using contexts such as time, location and device sensor information to model contextual services. The efforts related to providing high-level contexts to apps on smartphones have not seen the necessary advances in research required for adoption in spite of the rising popularity of smartphones.

1.1.1 Apps and the Cloud

The interactions between apps and the cloud has made it possible for apps to provide more information and features. This has led to the creation of Smart apps [12] and Intelligent Personal Assistants that can answer questions, make recommendations, and perform actions by delegating requests to a set of web services. These applications provide services based on the availability of contextual information related to the user. For example, Google Now [13] shows information about upcoming events it acquires by parsing emails from the users' Gmail address [14]. It is possible for Google Now to access and parse users' email since both the services exist within the same Google ecosystem. For other apps that are outside this ecosystem, the contextual information is not available without requiring some explicit steps from users to make such information available. This limited availability of information restricts the development of contextual services to only those apps that have access to a large dataset of information about the user, which is then analyzed and modeled into contextual information.

1.1.2 Limitations in Data Sharing

Mobile operating systems such as Android (v5.0.2) and iOS (v8.1.2) provide various ways for apps to share data, but lack a comprehensive framework to share contextual information across apps. Apps that wish to share contextual information must conform to a standard format for the data being exchanged that is understood uniformly by all involved apps. This places a burden on app developers to explicitly program interactions between different apps in order to facilitate the sharing of information though a mutually agreed API. The applications and their developers are thus unable to utilize all the contextual information available on a device, and are limited in the extent to which they can provide context-aware services.

The ranking of top apps is dominated by apps from developers Facebook, Google, Apple, Yahoo, Amazon and eBay. These six companies account for 9 of the top 10 most used apps, 16 of the top 25, and 24 of the top 50 [15]. This shows the preference of users to use apps within the same ecosystem which provides them with contextual features across apps developed by the same developer. This leads to other app developers prioritizing the integration of services and APIs from such popular apps the user is most likely to install. For example, the calendar app Sunrise [16] integrates birthdays and reminders from Facebook and Google+ along with a few others.[3] Information from only these services are synced and shown within Sunrise. Other comparable calendar apps that do not integrate these services can be deemed as being less attractive by the users based on the absence of features. This reduces the users' choice, and increases the pressure on app developers to integrate more services and APIs in their apps. The lack of a framework that supports the implicit sharing of information restricts developers to focus on a few services that are popular. This creates a necessity for apps to expose APIs to facilitate integration and cohesion which may help adoption by users. This leads to developers depending on APIs which are sometimes unsupported for interacting with an application that does not explicitly support integration. X-Callback-Url [17] is one such effort that provides documentation for services that can be integrated in other apps, but does not provide any way to structure or identify the data being shared.

1.1.3 Example Use Case: Movie Ticket Booking

In most common use cases, all the related information is available on a smartphone, but is distributed across different applications which are unable to share the information with each other. This creates difficulties in generation and consumption of contexts in

[3]As of January 01, 2015 Sunrise has the most number of connected services (16) integrated in a calendar app. A full list of services can be found at https://calendar.sunrise.am/

a smartphone, and forces the user to interact with different apps in order to complete a task covering a single context. This situation can be described using the movie ticket booking use case shown in Fig. 1.1, where the various steps taken by an user from booking the movie ticket to attending the movie show require the use of separate apps that do not share related information even though acting in the same context. The various steps followed by the user and the duplication of information and effort can be seen in the following steps:

1. **Movie Booking App**: The user enters or selects the movie's title, theater location, the show's date and time. The app generates the ticket and seat information which is stored within the app or sent to the user as an email or a text message.

2. **Calendar App**: The user creates an entry for the movie in the calendar. The title, date/time and location fields of this entry are duplicated by the user from the movie's title, show time and theater location. The user also has the option of adding a list of contacts who will be attending the movie with him/her.

3. **Messaging App**: The user forwards or copies the information containing the movie information sent by the movie booking app in a text message. The list of recipients is most likely related to the contacts added in the calendar. Here the user duplicates the movie information when entering the message contents, and the list of contacts as the recipients.

4. **Maps App**: The user uses the maps app to access route and navigational information when going to the theater, which requires entering the theater's address to set it as the destination. This information is duplicated along with the user being required to remember the location or look it up in a previously stored place such as the movie booking app or within messages. Some calendar apps offer navigational features within the app [18, 16], which require the user to open the calendar app in order to use this feature. Calendar apps that provide a navigational link in the notification only do so when the notification is displayed to the user. To access the navigational features at other times, the user needs to open the maps app or the calendar app containing a map.

5. **Accessing seat information**: At the theater, the user may require the seat numbers and ticket information to enter the theater or to print the tickets. This information can be accessed from the movie booking app, or in the copy stored as a message or an email, which involves several steps from opening the app to finding the relevant information. To make this information easily accessible at the theater, the user can use an app that offers reminders based on the location. The contents of such a reminder would include the required movie information,

and the trigger would be the theater's location. Setting up the reminder requires effort on the part of the user and further duplication of the movie information.

Table 1.1 shows the information entered by the user in various apps, with the fields that were entered multiple times being counted as being duplicated information. The label *APP* denotes information generated by the app, *USER* denotes information entered by the user for the first time, and *DUP* denotes duplicated information. It can be clearly seen from the table that a large amount of information is duplicated, which increases the effort required to complete the task as the user enters the same information in different apps. The overall user experience becomes disjointed as each app acts in an individual capacity based on the information available to it. This example shows the need for related information (in this case the movie's information) to be shared between apps to facilitate contextual services that will offer users a unified experience resulting from the availability of information across apps pertaining to the same context.

Table 1.1: Information acquired by various apps related to the movie ticket booking use case

App used	**Movie Title**	**Show Date/Time**	**Attending Contacts**	**Theater Location**	**Ticket Info**
Booking	*USER*	*USER*	*NA*	*USER*	*APP*
Calendar	*DUP*	*DUP*	*USER*	*DUP*	*DUP*
Messages	*DUP*	*DUP*	*DUP*	*DUP*	*DUP*
Maps	*DUP*	*DUP*	*DUP*	*DUP*	*DUP*
Reminder	*DUP*	*DUP*	*DUP*	*DUP*	*DUP*

1.2 Statement of the Problem

The problems and limitations described in the previous section can be mitigated with a Contextual Data Sharing Model that pervasively manages and mediates access to contexts on a smartphone. The model would allow for simple and intuitive access to contextual information stored across applications. The design and implementation of such a model is motivated by three key challenges in the area of context-awareness:

1. Identifying the contextual information;

2. An effective method for accessing this information;

3. Defining a contextual data store for storing the information.

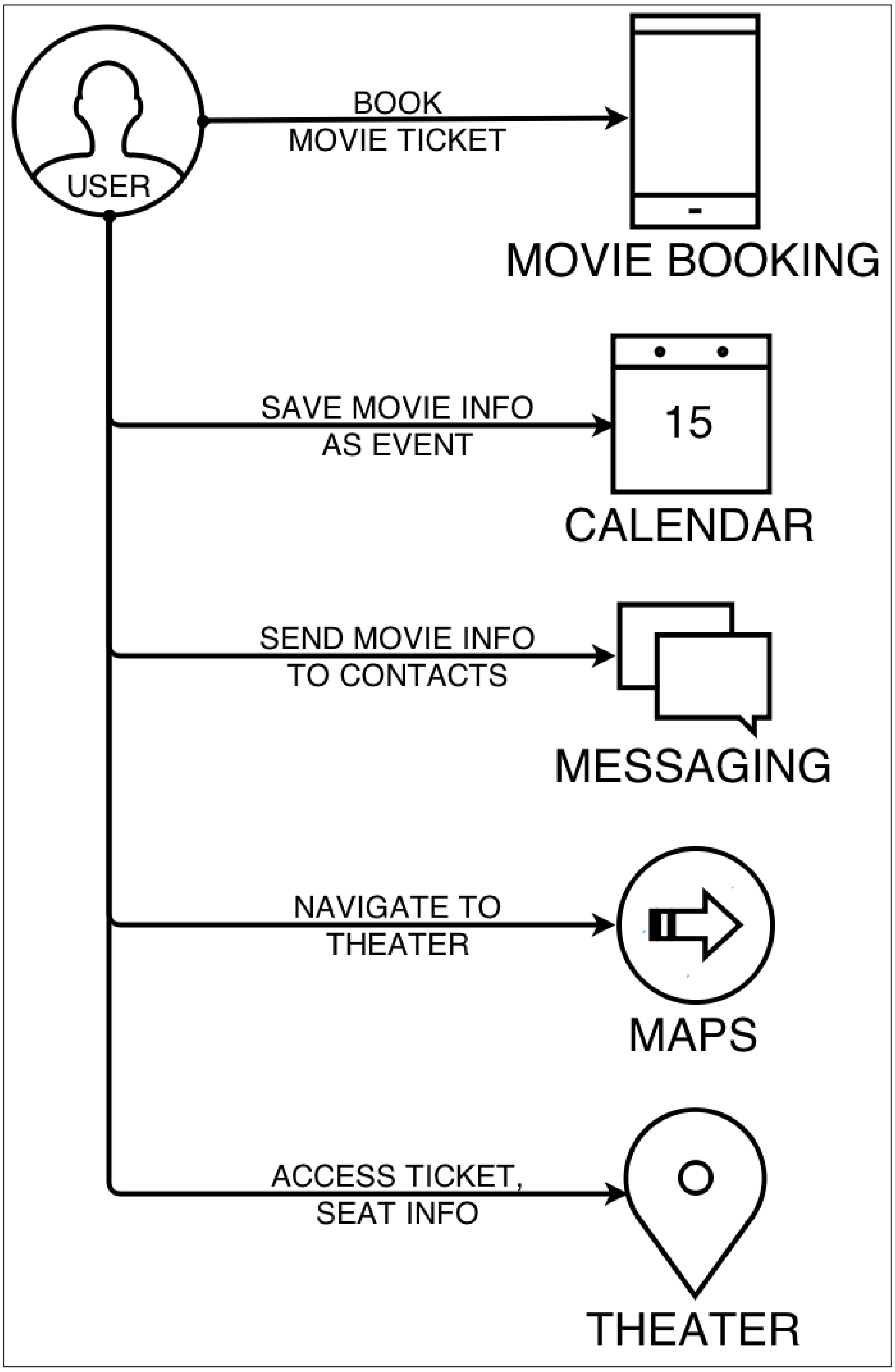


Figure 1.1: A use case highlighting the use of multiple apps used in the context of a movie ticket booking

The model allows apps to access contextual information without requiring explicit interaction and identifications of other apps, which leads to creation of features and services that help create a better user experience.

1.3 Purpose of Research

The purpose of this research is to design a Contextual Data Sharing Model and its various components in a manner that can be easily integrated and used by existing applications. The implementation of the model is a proof-of-concept demonstration that shows the viability of the model, and its impact on the user experience.

1.4 Significance of Research

The Contextual Data Sharing Model provides applications access to contextual information which can be easily stored and shared, and allows app developers to integrate contextual use cases in a simpler and more intuitive way. The contextual services can be designed relevant to the user's tasks without directly collaborating or sharing information with other apps. This allows a better user experience on the device, and leads to better features that allow recognizing and handling tasks the user is most likely to perform.

1.5 Primary Research Questions

The primary research questions that motivated the design and implementation of the Contextual Data Sharing Model are-

1. How can contextual information be structured in a uniform way?
2. How can contextual information be stored in a context datastore?
3. How can apps share contextual information through the Contextual Data Sharing Model without direct interaction or awareness of other apps?
4. Can an implementation of the model be created using a platform's native technologies?
5. What is the viability and performance of such an implementation?
6. Is the implementation stable and efficient to be used practically?
7. What are the impacts of the implementation on user experience?

The answers to these questions form the basis and motivation of this research.

1.6 Hypothesis

Primary - availability of contextual information leads to a better user experience

The primary hypothesis of this research is that the availability of contextual information for smartphone applications can lead to better features and an intuitive user experience. The hypothesis can be tested by comparing the user experience of use cases with and without the use of the Contextual Data Sharing Model. The conclusions reached through testing verify the impact of contextual information on user experience and app development.

Secondary - use of native technologies makes it easier to develop and manage contextual information

The secondary hypothesis of this research is that using native technologies in implementation allows for easier development and management of contextual information. The hypothesis states that using native technologies to implement the Contextual Data Sharing Model makes it easy to manage the information on a device and facilitates the sharing of information across applications without the need for developing or utilizing complex technologies that a platform does not natively support. It also makes it easy for developers to integrate contextual information and services into applications, which allows them to focus on developing contextual services rather than acquiring information.

1.7 Research Design

Different parts of the work follow different research designs. The representation of contexts is largely evolved from previous approaches and research in this area. The storage of contextual information is more considerate about the restrictions of a smartphone device, and is based on managing efficiency with performance. The implementation of the model is experimental in its approach as it attempts to combine the various components of the Contextual Data Sharing Model with the existing smartphone platform environment.

1.8 Assumptions, Limitations and Scope (Delimitations)

1.8.1 Assumptions

1. All apps involved or specified will use the Contextual Data Sharing Model in the specified manner.

2. The app that generates or identifies contextual information will correctly add it to the context data store.

3. Apps are aware of the contextual nature of the information acquired or entered by the user.

1.8.2 Limitations

The research is cognizant of the following limitations-

1. Apps need to adapt and use the Contextual Data Sharing Model in order for it to work across the device.

2. Apps can only provide contextual services based on the availability of information in the context datastore. When such information is absent, the app cannot provide contextual services.

3. The onus is on apps generating or identifying contextual information to insert it in the context datastore. If apps fail to add information to the datastore, the apps may not be able to provide contextual services.

4. The implementation of the model varies in some aspects depending on changes in platforms, devices and use cases.

1.8.3 Scope

The scope of the research is to make contextual information accessible to apps within the smartphone ecosystem through a framework that is designed with a bias towards popular smartphone operating systems like Android and iOS. The implementation of such a framework is based on the demonstration of the model on an unmodified version of Android with the aim to demonstrate the feasibility, impact and performance of the model. The implementation is termed as *working, but experimental*, and needs further efforts for testing and handling more use cases.

1.9 Summary

The Contextual Data Sharing Model enables apps to access contextual information without explicitly interacting with each other. This allows apps to create contextual services based on the availability of data which leads to the creation of better features and services that make the user experience richer and more engaging. The described movie ticket booking use case requires the user to enter the same information multiple times in different apps which duplicates information and increases effort. By using the

Contextual Data Sharing Model, the amount of effort and information entered by the user can be significantly reduced as apps share the information based on the context of the task implied by the user. The apps identify the nature of shared information and provide related contextual services accordingly. This allows features such as the calendar showing the movie event without requiring the user to enter any data. The maps app can provide routes to the theater based on the show time of the movie with a reminder at the theater showing the ticket and seat information. The use of contexts as defined in the model does not require dependence on the apps that generate or identify the information, which allows various apps to act independently.

1.10 Thesis Outline

This thesis is structured as follows:

Part II - Literature Review and Technical Background

Chapter 2 - Context-aware Computing. Chapter two looks at the previous approaches that identify and use contextual information. The emergence of context-aware computing and its use in mobile devices is also examined.

Chapter 3 - Data Sharing in Smartphone Applications. Chapter three discusses the various data sharing methods available to smartphone applications on Android and iOS.

Part III - The Contextual Data Sharing Model

Chapter 4 - Context Definition. Chapter four discusses the structuring of contextual information using Context Definitions. The chapter introduces the use of Context Definitions to store contextual information in a structured schema to provide a uniform representation across apps. The design and structure required for an implementation of the Context Definitions is also discussed.

Chapter 5 - Context Database. Chapter five introduces the Context Database used for storing contextual information. The design and structure of the Context Database along with its performance is also discussed.

Chapter 6 - Contextual Data Sharing Model. Chapter six introduces the Contextual Data Sharing Model and its components. The chapter discusses the various components and their responsibilities along with how contextual data sharing is achieved through the model. A demonstration of how apps would use the model is also discussed through a use case.

Part IV - Contextual Data Sharing in Android

Chapter 7 - Implementation. Chapter seven discusses the implementation of the Contextual Data Sharing Model on Android. The software and design approaches used in the implementation of the various components are also discussed. A demonstration of an use case with the apps using the Contextual Data Sharing Model on Android is also provided.

Chapter 8 - Performance Evaluation. Chapter eight discusses the various performance and user effort metrics of the implementation of the Contextual Data Sharing Model on Android. The focus of the evaluation is on the user effort, the operation times of the queries and the CPU utilization of the apps.

Part V - This Research and its Future Potential

Chapter 9 - Conclusion. Chapter nine presents an overview of the research, listing the advantages and validation of using the Contextual Data Sharing Model. Conclusions are drawn regarding the performance and user experience of the model along with its outcomes.

Chapter 10 - Future Work. Chapter ten discusses the various approaches that can extend and improve upon this research. The chapter provides various approaches to extend the model to other devices and platforms such as iOS, wearable computing, and smart devices. An idea of an ecosystem of devices where users are provided services based upon their contextual information is also discussed.

Part II

Literature Review and Technical Background

2

Context-aware Computing

"If I have seen further it is by standing on the shoulders of giants."
– *Isaac Newton*

2.1 Defining Context

The word *context*, derived from Latin *con* meaning with or together, and *textere* meaning to weave, denotes context as a profile and an active process dealing with the way humans weave their experiences within their whole environment to give it meaning. The word 'context' also denotes the study of human 'text' and the idea of 'situated cognition'. The idea that context changes the interpretation of text goes back many thousand years.

In computer science, context awareness refers to the idea that computers can both sense, and react based on their environment. Context aware devices try to react based on rules and intelligent stimulus based on the user's current situation. The term context awareness in ubiquitous computing was first defined in 1994 as: *"software that adapts according to its location of use, the collection of nearby people and objects, as well as changes to those objects over time."* [19, 20] Previous research in this area has resulted in several adaptive and personalized applications based on the notion of user profile and context [21]. Some of them that have motivated and influenced this research are summarized in this section.

The definition of context as given by Dey [22] is: "*Context is any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and applications themselves.*" A definition of context-aware systems is also provided as: "*A system is context-aware if it uses context to* *provide relevant information and/or services to the user, where relevancy depends on the user's task.*" Their paper discusses the different ways context can be used by context-aware applications. Three categories of features that a context-aware application can support are given as: presentation of information and services to the user, automatic execution of a service for a user, and tagging of context to information to support later retrieval.

Zimmermann et al. [23] extend the context definitions with the idea of defining the task itself as part of the context since it characterizes the situation of the user. This central role of task in context is shared by Crowley et al. [24] and Kofod-Petersen et al. [25], who assume that the user's actions are generally identified by a set of tasks (actions) and are goal driven. Henricksen [26] gives more importance to task in her definition of context, which is: "*The context of a task is the set of circumstances surrounding it that are potentially of relevance to its completion*". Dey et al. [27] extend their definition of context with the statement *"Context is typically the location, identity and state of people, groups and computational and physical objects."* Abowd et al. [28] discuss how context can be considered a part of the process along with the state in which the users are involved. Chen [29] uses location as a context-providing parameter with activities and tasks taking place in a location.

2.2 Context in Mobile Devices

Battestini et al. [30] discuss how mobile phones can be used to create intelligent applications that are able to understand user needs through context and have the ability to adapt and provide recommendations. They extend the idea of a statistical approach to context recognition by defining contexts as clusters in the data [31], which can be used to recognize clusters unambiguously associated with high-level contexts or situations. Two examples of context-aware applications given are the Context Watcher and the Family Maps, built using the MobiLife[1] architecture and which utilizes various sources of information to predict and infer the users' needs. The authors express the need for a user-centric privacy and trust framework that will allow flexible information exchange while controlling the access to user data.

Malik and Mahmud [32] identified the following challenges for middleware that addresses context awareness:

1. Context acquisition to collect the items of the context: context-aware middleware can centralize context data from various sources and sensors.

[1]MobiLife is a user-centric architecture that aims to provide communication and sharing of items in order to manage complex lifestyles. *http://www.ist-mobilife.org*

2. Context representation that provides an efficient structure for retrieving context. The authors mention the various approaches for representing context. [33]

3. Context storage which stores correctly represented context in a structured and persistent manner.

4. Context interpretation based on different strategies and fields of research such as machine learning or complex event processing to enable context-awareness.

5. Context adaptation to use the context after interpretation in context-aware scenarios.

Yau et al. [34] describe RCSM, a system that creates ad hoc communication between devices to facilitate information exchange. They present two categories of middleware in pervasive computing based on interaction between devices or entities. Their implementation uses various sensors to detect light, noise, etc. in a classroom to trigger communication activity between students and instructor. The system uses a state-trigger based scenario where context states trigger activities.

Klein et al. [35] describe a context management architecture based on the producer-consumer role model and designed to acquire, manage, and distribute context information and to control the context quality. The architecture consists of a Context Provider where context information originates from and is provided to other entities of the architecture; a Context Broker that acts as a middleman to perform lookups and resolutions; and Context Consumers that use context data as an input for providing functionality. The authors note the the dependence of the quality of context information on connectivity and its impact on delay, accuracy, relevance, and confidence, which requires a Context Quality Enabler incorporated into the architecture to control the provision of context information.

Korpipaa et al. [36] describe a framework for the development of context-aware mobile applications that manages raw contextual information gained from multiple sources and enables higher-level context abstractions. The framework contains a central node that provides context information to clients through direct querying, subscribing to context change notification services or higher-level composite contexts managed by the context manager. Each context is described using six properties where each context expression contains at least one type and value, which together form a verbal description of the context.

Alidin and Crestani [37] describe the “just-in-time” approach where relevant information is retrieved without the user requesting it. This reduces the users effort, time and interaction and presents the relevant information to the user in the right time and place. The authors note how smartphones can react and adapt to the context to minimize user interaction and use context as information triggers to pro-actively present information to the user. This is also known as Just-In-Time Information Access system [38]. In this system, if the information is no longer relevant to the user interaction, then it is not considered to be a part of the context. The authors describe how some researchers incorporate too many dimensions of context which make the context models too complex to be implemented in smartphones. But if there are fewer dimensions of context, it can lead to context models being unable to understand the whole context. In their approach, they define 10 dimensions of context including time, location, sound and user profile. The context model has structured levels of context where sensor data is at the bottom level. Context dimensions or low-level context are characterized by interpreting multiple sensor data. High-level contexts are generated from multiple context dimensions and describe the user's current context. A user scenario depicts the situations encountered by the user and is made up of one or more high-level context. The system works by monitoring context, predicting user information needs in any given context and pro-actively providing the user with relevant information with the aim to reduce user interaction. The system infers user information need in context based on the user's current context and an information need analyzer. If this information is not available, the system predicts user information need by acquiring context from other closely similar users, or by utilizing the user's location, preference, and context.

Falcarin et al. present an architectural framework for context-aware services called *Context Data Management* [39] that provides interoperability and domain independence for third-party context-aware applications. The authors describe a high-level framework that provides a set of defined roles for abstract components that offer APIs. The framework provides context data management, context analysis and integration of context in mobile devices, with a focus on sensor data. The main component called Context Broker manages communications with the other components, while Context Providers store information that can be queried by Context Broker or Context Consumers. Context Sources provide data to the Context Broker through asynchronous communication and are typically located on mobile devices. The Context Broker organizes data into different Context Scopes, which are subscribed by Context Consumers in a publisher-subscriber model. A Context Cache stores recent context data, which is moved to a Context History database upon expiry.

Context Directory [40] is a framework that helps mobile applications to achieve context-awareness through context models, interpretation methods and adaptation possibilities. The framework consists of context clients, which communicate with the sensors on the mobile devices to collect context attributes for creating context representations. It uses a key/value model for simplicity and to allow multiple context clients to collect context attributes on different devices which can be merged into context directories. The communication between directories and clients is handled by the context directory protocol. A context-aware API allows development of context-aware applications for specific mobile platforms and can be used to build a context-aware application based on the interface. Developers can directly interact with context clients for more control over the sensitivity of contexts. The context is interpreted using a variety of algorithms based on usability of context-aware scenarios. A demonstration of the framework is shown through a context-aware application for Android, where the context-aware API was modeled to be similar to the Android API. Context commands, which describe the adaptation of user interfaces and the behavior of executed commands are developed based on the context. Contextual information is implemented by a context-matching module that sorts information based on time and relevancy. The authors note the possibility and complexity of using automatic contextual reconfiguration using Dalvik's reflection mechanism.[2] The application model and possible actions need to be registered in the context directory for executing context-triggered actions.

Most definitions and implementations of context-aware systems focus on using aspects such as time, location and sensor information to model contextual systems. While these can be used to cover a large variety of common use cases in mobile devices, there is more information associated with other contexts on mobile devices that needs to be recognized and used.

2.3 Context Frameworks utilizing the Cloud

Many contextual models use a cloud-based approach, where the cloud is utilized to offer services not possible on a mobile device and to share information between multiple devices. Offloading work to the cloud enables services not previously possible on mobile devices as described in [41, 42, 43, 44, 45].

One such approach related to this research is *COSMOS* [46], which describes a cloud-based PaaS (Platform as a Service) system that provides infrastructure for mobile apps to share data. The authors emphasize the incentive for mobile apps to share information with one another on a large scale through a service based in the cloud and hosting the mobile apps' datasets. They provide an implementation model that hosts app data in the cloud and provides seamless experience by sharing that data with multiple apps. The COSMOS PaaS system contains Sharing Middleware (SMILE) that mediates sharing between mobile tenants and the COSMOS data store. The implementation focuses on sharing data across apps hosted in COSMOS and targets user services towards contextual information gathered from various datasets. An example provided is that of a user going to a conference, where his conference date and location is used

[2]Dalvik is the process virtual machine (VM) in Android and executes applications written for Android. Reflection is the ability of a computer program to examine and modify the structure and behavior (specifically the values, meta-data, properties and functions) of the program at runtime.

to book airline tickets and the hotel room. The COSMOS datasets provide all the information required without the user specifying these requirements. For all services to work, the app must be hosted in COSMOS and must use its architecture.

The Cloud Personal Assistant [47] is a cloud service that manages the access of mobile clients to cloud services. It provides service discovery and invocation, and stores the results and history for delivery to the mobile client. The assistant receives a set of tasks to execute, and returns the results when the mobile client needs them. There are three tiers called the user tier, the task tier and the service tier. The framework provides independence from connectivity as the results are stored in the cloud and returned when the mobile client requests them.

2.4 Classification of Context-aware Systems and Services

A surveys and comparison of context-aware systems and models is presented in [21]. Baldauf and Dustdar [48] survey various context-aware middleware and frameworks and present a comparative analysis focusing on the context services. They conclude their analysis with two points, the first being that appliances should vanish into the background to make the user and his tasks the central focus rather than computing devices and technical issues. The second is that context-aware systems are able to adapt their operations to the current context without explicit user intervention and thus aim at increasing usability and effectiveness by taking environmental context into account.

According to Strang et al. [33] there are six ways of modeling and representing context data amongst existing works:

1. Key-value pairs, which are primitive and cannot handle complex context information, but easy to integrate;

2. Graphical models, which describe the structure of context, but do not separate the data layer from code layer;

3. Object-oriented models, which encapsulate the context in objects and can provide special interfaces, but are not efficient at large-scales;

4. Logic-based models, which define context as a set of facts and evaluative expressions which can be used to derive new facts, but are complex and restrictive;

5. Ontology-based models, which use formally specify concepts and interrelations of the human language, and can grow to a large size;

6. XML models, which are hierarchal data structures made up of XML tags with attributes and contents and offer dynamicity of information, but are require separate services to interact with the data.

Schilit et al. [19] defined four categories of context-aware applications that overlap with the categories defined by Pascoe [49], who defined the following five categories of context-aware applications:

1. Contextual information or proximate selection or contextual sensing, which describes the supply of context-aware content by an application. This is a form of matching or rating of information by context.

2. Contextual commands, which are the category of applications that change their presentation or execution flow based on the context.

3. Automatic contextual reconfiguration, which exchanges parts of the software based on context.

4. Context-triggered actions that enable executing applications and operations without inputs based on the context. Similar mechanisms are used by online shopping systems.

5. Contextual augmentation, which enhance the perspective on the environment by adding additional information to the reality.

Chihani and Bertin [11] give a new approach for classifying context-aware communication systems, where adaptation is performed based on how context is used. They identify services as Instant or Deferred and On Device or On Cloud based on their implementation instead of their functionalities. They also note that the most used context sources are physical information such as location and time, environmental information such as weather, personal information such as health, mood and social information such as relationships and applicative information such as emails. They discuss how high-level knowledge can be derived from raw contextual information to give a better understanding of the user.

2.5 Context Representations

There have been various ways proposed to represent context. Most researchers represent context using formats tied to their particular approach. Some of the commonly used and popular formats for representing contexts are Resource Description Framework (RDF) [50] and OWL [51].

2.5.1 Resource Description Framework (RDF)

RDF is a data model that uses statements about resources in the form of subject-predicate-object expressions known as *triples*. The subject denotes the resource, the predicate denotes the traits or aspects of the resource and expresses a relationship between the subject and the object. For example, for representing the English language statement *'New York has the postal abbreviation NY'* in RDF, *'New York'* would be the subject, *'has the postal abbreviation'* would be the predicate, and *'NY'* would be the object. To be encoded as a triple, the subject and predicate need to be expressed as URI resources, and the object can be a resource or a literal element. The example expressed in Listing 2.1 is in N-Triples form, where *'urn:x-states:New%20York'* is the URI for a resource that denotes the US state New York, *'http://purl.org/dc/terms/alternative'* is the URI for this predicate[3], and *'NY'* is a literal string.

```
<urn:x-states:New%20York>
<http://purl.org/dc/terms/alternative>
"NY"
```

Listing 2.1: Schema for *Event* Context

2.5.2 Web Ontology Language (OWL)

The Web Ontology Language (OWL) is a family of knowledge representation languages or ontology languages for authoring ontologies or knowledge bases. The languages are characterized by formal semantics and RDF/XML-based serializations for the Semantic Web. The data described by an ontology in the OWL family is interpreted as a set of *individuals* and a set of *property assertions* which relate these individuals to each other. An ontology consists of a set of axioms which place constraints on sets of individuals (called *classes*) and the types of relationships permitted between them. These axioms provide semantics by allowing systems to infer additional information based on the data explicitly provided.

Languages in the OWL family are capable of creating classes, properties, defining instances and its operations. An instance is an object and corresponds to a description of an individual logic. A class is a collection of objects and corresponds to a description of a logic concept. A class may contain individuals and any number instances of the class. An instance may belong to none, one or more classes. A class may be a subclass of another, inheriting characteristics from its parent superclass. All classes are subclasses of *owl:Thing*, the root class. All classes are subclassed by *owl:Nothing*, the empty class. No instances are members of *owl:Nothing*. Modelers use *owl:Thing* and *owl:Nothing*

[3] An alternative title with a human-readable definition is available at http://dublincore.org/documents/dcmi-terms/index.shtml

to assert facts about all or no instances. A property is a directed binary relation that specifies class characteristics. It corresponds to a description logic role. They are attributes of instances and sometimes act as data values or link to other instances. Properties may possess logical capabilities such as being transitive, symmetric, inverse and functional. Properties may also have domains and ranges.

Context representation in smartphones

Knowledge graphs and collective data banks utilize RDF and OWL or their related formats to store contextual data and its relations. Web services utilize different formats such SOAP, WSDL, UDDI [52] or JSON [53] to communicate data. All of these forms are non-native on smartphone platforms and require parsing before the data can be utilized or manipulated. This places limitations on the utilization of models that use these formats for interacting with context since smartphone applications are developed using different technologies which makes it difficult to integrate traditional context representations into the application design.

2.6 Smart Apps

The word *Smart Apps* is derived from the word *smartphone*, which stands for a mobile phone with an operating system that offers features such as a personal digital assistant, a digital camera, a media player, and/or a GPS navigation unit. The word *smart* in *smart apps* is used to denote the capability of an app to help users with the completion of their task by utilizing contexts such as location, time, sensor information or offering recommendations and automation which allow the task to be completed in lesser time and/or fewer steps [12]. The features and design of smart apps depend on the availability of contextual information. An example of a smart app is Fantastical [18], which allows users to type entries in natural language in a single text box to create events in the calendar instead of interacting with multiple entry fields and UI elements.

2.7 Intelligent Personal Assistants

An intelligent personal assistant is a mobile software agent that can perform tasks or services for an individual. These tasks or services are based on user input, location awareness, and the ability to access information from a variety of online sources (such as weather or traffic conditions, news, stock prices, user schedules, retail prices, etc.). Intelligent personal assistant technology is achieved through the combination of mobile devices, application programming interfaces (APIs), and the proliferation of mobile apps. An intelligent personal assistant can be designed to perform specific, one-time tasks specified by user voice instructions, or to perform ongoing tasks autonomously. One of the key aspects of an intelligent personal assistant is its ability to organize and maintain information such as emails, calendar events, files, and to-do lists.

2.7.1 Siri

Siri [54] is an intelligent personal assistant and knowledge navigator developed by Apple for its iOS platform. The application uses a natural language user interface to answer questions, make recommendations, and perform actions by delegating requests to a set of web services. The software adapts to the user's language usage and searches with use, and returns results that are individualized. Siri allows users to make reservations at specific restaurants, buy movie tickets or get a cab by dictating instructions in natural language. The application integrates with default iOS functionality, such as contacts, calendars and text messages and supports services from providers such as Google, Bing, Yahoo, Wolfram Alpha, Google Maps, Yelp! and Wikipedia. Siri also contains numerous pre-programmed responses to conversational and amusing questions.

2.7.2 Google Now

Google Now [13] is an intelligent personal assistant developed by Google. It is available within the Google Search mobile application for Android, iOS, and the Google Chrome web browser on personal computers. Google Now uses a natural language user interface to answer questions, make recommendations, and perform actions by delegating requests to a set of web services. Along with answering user-initiated queries, Google Now pro-actively predicts information the user will want based on their search habits, and by utilizing data from users' other Google services. Google Now is implemented as an aspect of the Google Search application and recognizes repeated actions such as common locations, repeated calendar appointments, and search queries a user performs on the device to display relevant information to the user in the form of 'cards'. The application system leverages Google's Knowledge Graph project[4] to assemble more detailed search results by analyzing their semantic meanings and connections.

2.7.3 Cortana

Cortana [55] is an intelligent personal assistant for Windows Phone 8.1 and the Microsoft Band. Cortana's features include being able to set reminders, recognize natural voice without the user having to input a predefined series of commands, and answer questions using information such as current weather and traffic conditions, sports scores,

[4]The Knowledge Graph is a knowledge base used by Google to enhance its search engine's search results with semantic-search information gathered from a wide variety of sources. More information can be found at `http://www.google.ie/insidesearch/features/search/knowledge.html`

and biographies from Bing. The 'Notebook' is a data collection unit where personal information such as interests, location data, reminders, and contacts are stored for access. Cortana uses this data in order to learn the user's specific patterns and behaviors and can add information as it learns. The user can view this information and specify what information is collected, which offers greater control over privacy settings compared to other assistants. Users can also delete information from the 'Notebook' if they deem it undesirable for Cortana to know.

2.7.4 Data Sources and Mode of Operation

Intelligent personal assistants work on the principle of analyzing and interpreting data to provide contextually relevant information and services. In order to replicate the functionality of personal assistants, an app needs access to data and resources which it can utilize to formulate the contextual responses to the users' queries. In the case of Google Now, it has access to data aggregated from all of Google's diverse services, and needs the user to actively use those services in order to function as intended. Siri does not provide personalized recommendations beyond those obtained from data limited to some services on the device. It however, uses natural language processing based in the cloud to identify the users requests and to provide appropriate responses. Similarly, Cortana utilizes data acquired from the device and Microsoft's services in order to provide functionality similar to Google Now.

Developing features in apps comparable to personal assistants requires access and analysis of data related to the user's context, whose availability is limited to a developer. This can be seen through the difference between the advent of smart apps and personal assistants, where smart apps are developed based on using functionality and comparably lesser data. Making such data available to developers is not possible due to concerns about security and privacy, which do not exist for personal assistants as they utilize data within a single ecosystem or tightly coupled ecosystems.

3

Data Sharing in Smartphone Applications

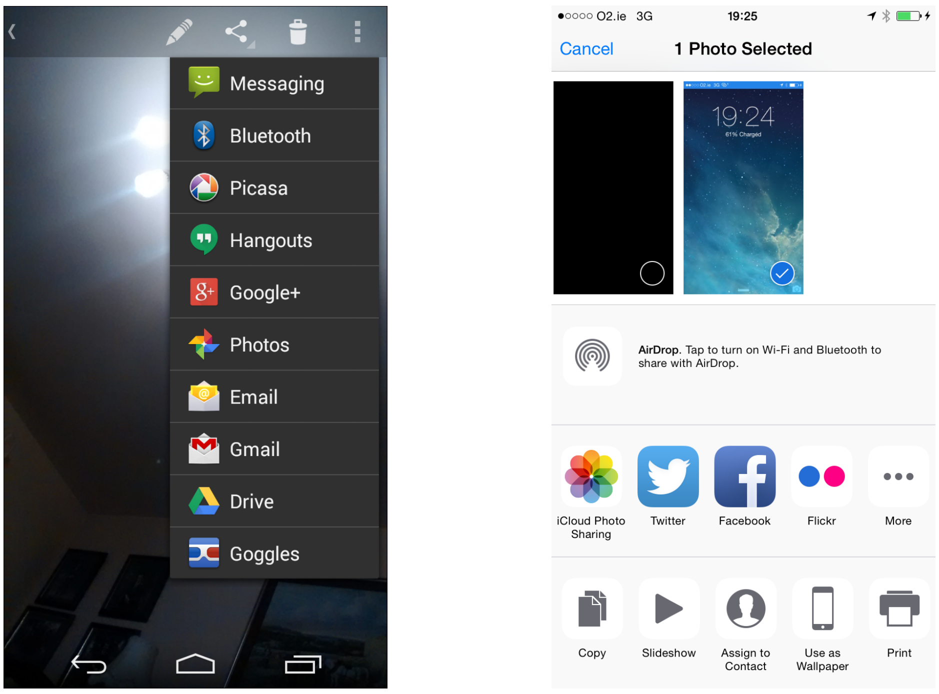
"Data! Data! Data! I can't make bricks without clay!"
– Sir Arthur Conan Doyle

Data sharing can be defined as the ability to share the same data resource with multiple applications or users (collectively called clients) [56]. It implies that the data is stored or generated in one or more applications on a device and there is some framework or middleware that allows it to be shared between applications.

Unlike applications on traditional operating systems, smartphone applications have a much more restrictive security model called *sandboxing*. Sandboxing is a security mechanism that separates applications from each other and restricts any unspecified access between them. This prevents one application from corrupting or over-writing another application's data. This results in aggressive restrictions that limit interactions with other applications. Data sharing is possible only through the use of APIs provided by the system. The following sections discuss the various data sharing mechanisms available in iOS and Android. Apart from the data sharing methods described below, applications cannot utilize or define other APIs, and neither can they directly access data from another app due to the restrictions placed by the sandboxing model.

3.1 Sharing Common Data Types

Both iOS and Android support sharing of common data types such as text and images through a dedicated *share* menu that can send data objects from within an app to other apps. On Android, this is achieved through the Intent and Intent-filter mechanism [57], whereas on iOS, the same is achieved through UIActivityViewController [58]. Both platforms allow the selected data object to be shared with all applications that have registered the capability to handle the object's MIME type [59]. The destination of the target app is selected by the system or by the user themselves though a UI element as seen in Fig. 3.1.



(a) Android 4.4 (b) iOS 8

Figure 3.1: Sharing data objects using an app's *Share* menu

The system populates the sharing list with applications that have explicitly declared the capability to handle the type of context being shared. Applications declaring the ability to handle data objects of type *image* are displayed in the list shown when an image is selected and shared. In Android, the explicit declaration is mentioned through the applications' *manifest* [60] file, whereas on iOS it is declared programmatically via code.

The system provides the required APIs for handling and sharing commonly used data types such as image, audio, web-pages, and text. Any other data type must be explicitly known and uniformly interpreted by all the apps that want to share data objects of that type. For example, a calendar event can be considered a complex data type available through the system's calendar API. Events such as a restaurant booking or a movie show are similar to a calendar event, but are not recognized data types. Apps that wish to declare or share these data types cannot guarantee that other apps will correctly recognize and interpret them.

3.2 Custom URL/URI schemes

In computing, a Uniform Resource Locator (URL) is a subset of the Uniform Resource Identifier (URI) that specifies where an identified resource is available and the mechanism for retrieving it. The scheme name (or protocol) of a URL is the first part of a URL. For web pages, the scheme is usually *http* or *https*. iOS and Android support some URL schemes related to web-pages, telephony and messaging by default [61, 62]. Applications can specify their own custom URL scheme such as *myapp://something* which is resolved by the system to open the particular functionality within an app associated with the scheme. This allows other apps to communicate with an app through a protocol defined by its URL/URI scheme.

The resolution of URL/URI links is done by the system, which identifies the correct app associated with the scheme and opens it. If there is more than one app associated with a scheme, the system shows a dialog to select an application similar to the dialog shown in the *share* menu. The system restricts the resolution of some schemes to fixed applications, which cannot be changed. If some user application declares a URL/URI that is identical to a restricted scheme, the default system app is launched instead of the user app. URL handling on Android devices works through the Intents mechanism, where apps register to get launched in response to certain specified actions.

Applications can use custom URL/URI schemes to directly navigate to activities within an app, or to send and receive data. Small amounts of data can be easily encoded into the URL/URI in a way similar to how websites are accessed, though the system does not require network connectivity to resolve these schemes. Applications can behave differently based on the data passed in the URL. For example, the Map app can be sent location coordinates as parameters in the URL to open an activity displaying the location specified by the coordinates on the map. The app that generates the URL/URI needs to be aware of the correct syntax the Map app requires. In general developers are required to know the various schemes and syntaxes required to integrate popular services from other apps. Some developers have created a public library of custom URL/URI schemes [8] supported by various apps as a way for other developers to discover services easily. Using such libraries still requires the developers to be explicitly aware of the exact URL/URI scheme required by the target app in order to interact with it.

Deep Linking

In the context of the Web, deep linking consists of using a hyperlink that links to a specific generally searchable or indexed piece of web content on a website. For example using *http://example.com/path/page* to navigate to a particular page rather than the home page at *http://example.com/*. In mobile apps, deep linking consists of using a URI that links to a specific location within a mobile app (*exampleApp://location/123456*), rather than simply launching the app (*exampleApp://*).

The greatest benefit of mobile deep linking is the ability for marketers and app developers to bring users directly into the specific location within their app with a dedicated link. Unlike deep links on the web, where the link format is standardized based on HTTP guidelines, mobile deep links do not follow a consistent format. This causes confusion because different sets of links are required to access the same app on different platforms.

Alternate solutions developed include one approach where a smarter deep link is created that triggers the most appropriate response depending on the device being used. A solution developed by AppsFlyer called OneLink [63] detects the device type and the installed apps and triggers a Web or Mobile deep link or opens the appropriate App Store in case the requested app is not installed on the user device, also known as deferred deeplinking. Another solution proposed by URX called Omnilinks [64] requires prefixing a web link with `urx.io/` to convert it into a deep link, which works across all devices. This routes users into a specific page in an app if that user has the app installed. App Links [65] is a deep linking standard developed by Facebook that makes it possible to launch an app containing content shared on Facebook (or another App Links-enabled app). Quixey created AppURL [66] which is an open standard for deep linking across platforms and also allows search engines to crawl in-app links. Quixey also produces search algorithms for in-app searches that provides results in the form of deep links. For example, if the user searches for "Mexican food", the user is provided with results from apps that provide restaurant services such as Yelp, GrubHub, OpenTable, and Foursquare, where the results act like a shortcut that take the user directly into that page in the app if it is installed.

Google provides App indexing [67], which allows developers to connect pages from their website with specific content within their smartphone app on Android. This enables smartphone users who have the app installed to open it directly from relevant mobile search results on Google. Developers can make it possible for Google Search to open specific content in their app by providing intent filters for relevant activities. This requires configuring both the app and the website along with a description of how they are related to to oder to show *Open in app* deep links in search results. This feature is limited to apps that also have a website that Google can search and index. Currently, only Google search can make use of app indexing. Such kinds of contextual information sharing is not available to all applications in general.

3.3 Telephony and Messaging data

Applications can share data inherently through the use of various options made available by system. Commonly required data such as Calendar and Contacts are provided as part of standard APIs that are made available to developers. *EventKit* [68] on iOS provides developers access to calendar and reminders, which lets applications interact with the user's calendars and reminders. For accessing contacts, iOS has *Address Book* [69] that provides a centralized database of the user's contacts that any application can access and update. Android features Content Providers [70] that act like a centralized database providing access to other apps through the use of specific APIs. Data such as contacts and calendars are made available to other apps by exposing their content provider API. Every version of Android includes these two Content Providers by default, and others can be added by apps when they are installed on a device. Fig. 3.2 shows the Calendar Provider which is a Content Provider that provides access to calendar data on Android. Accessing Content Providers for user apps requires permission and access to the Content Provider's URL/URI. There is no abstract mechanism whereby apps can discover and/or connect to Content Providers already present on the device.

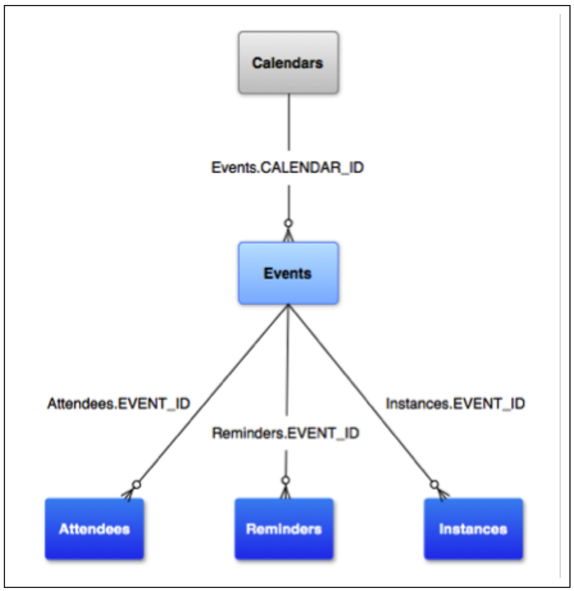


Figure 3.2: Calendar (Content) Provider in Android

3.4 Comparison with Traditional Data Sharing Methods

Android and iOS, despite being based on or closely related to POSIX based systems[1] do not allow apps to use traditional data sharing mechanisms such as file-copy, pipes or data dumps.

3.5 Document Pickers

A document picker allows an application to request selection of data objects through another app. Common uses include selecting image files or contacts within an app that does not have access to these resources. Android uses the Storage Access Framework (SAF) [71] that allows users to browse and open documents, images, and other files across all of their their preferred document storage providers. iOS has a similar mechanism known as Document Picker [72] that lets users select documents from outside an apps sandbox.

3.6 Methods exclusive to iOS

3.6.1 Pasteboard

iOS has a common data dump, called UIPasteBoard [73] which acts like a clipboard sharing service. Apps can place data on the pasteboard that is accessible globally. Pasteboards can be given unique identifiers that can be shared between apps. The data on a pasteboard persists even if the app that used it is terminated. Pasteboards are flexible in the size of data to be shared. However, any applications accessing the pasteboard can overwrite or change its contents.

3.6.2 Airdrop

AirDrop [74] is used to share photos, documents, URLs, and other types of data with apps and nearby devices. It uses peer-to-peer networking to find nearby devices and connect to them.

3.6.3 Shared Keychain

A keychain [75] is an encrypted container that holds passwords for multiple applications and secure services, used by iOS as a password management system [7]. Applications

[1]POSIX is an acronym for Portable Operating System Interface, and defines the application programming interface (API) and utility interfaces for software compatibility with variants of Unix and other operating systems. Android uses the Linux kernel, while iOS uses Darwin, both of which are loosely based on POSIX systems.

can store data securely to a shared keychain which is accessible only to other applications using the same app ID prefix. The amount of data stored in the keychain is limited as it is not suitable for storing large amounts of data.

3.6.4 iCloud - Shared Storage

Developers can use the iCloud service [76] to store large amounts of data, which can be accessed by apps having the same ID prefix. Therefore, apps not having the same ID prefix and looking to share data must use third-party libraries or frameworks which will allow them to share data with other apps that agree to use the same framework.

3.7 Methods exclusive to Android

3.7.1 Intent and Intent Filters

An Intent [57] is an abstract description or an *intention* of the operation to be performed. Intents are asynchronous messaging objects used to request functionality from other app components, which allows an app component to interact with other components within the same application as well as with components in other applications. An Intent provides a facility for performing late runtime binding between the code in different applications. Its most significant use is in the launching of activities, where it can be thought of as the glue between activities.

Intents are generally used to launch Activities,[2] to interact with services, and to send broadcast messages. Android supports explicit and implicit intents. An application can define the target component directly in the intent (explicit intent) or ask the Android system to evaluate registered components based on the intent data (implicit intents).

Explicit Intents

Explicit intents explicitly define the component which should be called by the Android system, by using the Java class as identifier. Explicit intents specify the component to start by its fully-qualified name.[3] Listing 3.1 shows a function in *app1* that starts an activity in *app2*. Without the fully qualified name and necessary permissions, it is not possible to start or interact with activities from other apps. When an explicit intent is

[2]An Activity is an application component that provides a screen with which users can interact in order to do something, such as dial a number, take a photo, send an email, or view a map. Each activity is given a window in which to draw its user interface. The window typically fills the screen, but may be smaller than the screen and float on top of other windows.

[3]In Android, each app is defined by a unique package namespace. In order to start another app's activities, the app's package and the activities' name must be known. Also, the app must declare the necessary permissions to let another app start its activities.

used to start an activity or service, the system immediately starts the app component specified in the Intent object.

```java
package msc.prototype.demo.app1;
someFunction() {
  Intent intent = new Intent(this, "msc.prototype.demo.app2.Activity);
  intent.putExtra("Value1", "This value one for ActivityTwo ");
  intent.putExtra("Value2", "This value two ActivityTwo");
  startActivity(intent);
}
```

Listing 3.1: Starting an activity in another app by using an explicit intent

Implicit Intents

Implicit intents declare a general action to perform which is handled by a component from any app that has declared the capability to perform the requested action. For example, to show the user a location on a map, an implicit intent can be used to request another app to show a specified location on a map. When using an implicit intent, the Android system finds the appropriate component to start by comparing the contents of the intent to the intent filters declared in the manifest file of other apps on the device. If the intent matches an intent filter, the system starts that component and delivers it the intent object. If multiple intent filters are compatible, the system displays a dialog so the user can pick which app to use. This is also how a share menu is populated by the system. Fig. 3.3 shows how an implicit intent is handled by the system to open another application. The *onCreate* method is used to open an application's activity and perform initialization operations. Listing 3.2 shows an example of an implicit intent, used to send `ACTION_SEND` text `MIME type` `PLAIN_TEXT_TYPE` to other apps capable of handling text.

```java
Intent intent = new Intent();
sendIntent.setAction(Intent.ACTION_SEND);
sendIntent.putExtra(Intent.EXTRA_TEXT, textMessage);
sendIntent.setType(HTTP.PLAIN_TEXT_TYPE);
startActivity(intent);
```

Listing 3.2: Sending contents to other apps by using an implicit intent

Intent Filters

An Intent Filter specifies the type of intents that an app component wants to respond to. An intent filter declares the capabilities of its parent component such as what it can do and what types of broadcasts a receiver can handle. It opens the component to receiving intents of the advertised type, while filtering out those that are not meaningful for the component. For example, declaring an intent filter for handling text shows the app in the share menu whenever some text is selected. An intent filter is declared in the app's manifest. If an activity does not contain an intent filter, it can be started only with an explicit intent. Listing 3.3 shows an intent filter that tells the Android system to launch the given activity launching the app.

```xml
<activity android:name="MainActivity">
    <!-- This activity is the main entry, should appear in app launcher -->
    <intent-filter>
        <action android:name="android.intent.action.MAIN" />
        <category android:name="android.intent.category.LAUNCHER" />
    </intent-filter>
</activity>
```

Listing 3.3: Declaring an intent filter that opens the activity upon app launch

Intent Data

An Intent object carries information that the Android system uses to determine which component to start (such as the exact component name or component category that should receive the intent), and information that the recipient component uses in order to properly perform the action (such as the action to take and the data to act upon). An intent primarily contains the name of the component to start, the action to perform, a URI object that references the data or the MIME type of the data, a category about the kind of component required to handle the intent, flags that define the metadata and extras, which are key-value pairs containing additional information required to complete the action. Intent objects can be used to marshal data across apps and processes, provided that the apps at both side of the connection are aware of the exact type and contents of the data object.

3.7.2 Broadcast Receivers

Broadcast Receivers [77] respond to broadcast messages (also called events or intents) from other applications or from the system itself. For example, applications can initiate broadcasts to let other applications know that some data has been downloaded to the device and is available for them to use. The broadcast receivers declared in the apps will intercept this communication and will initiate appropriate action. The broadcast receivers are declared by an app in its manifest, and must contain the details of the type of messages they wish to receive. To receive updates or information from another app's broadcasts, the app must know and declare the correct broadcast identifier in its manifest.

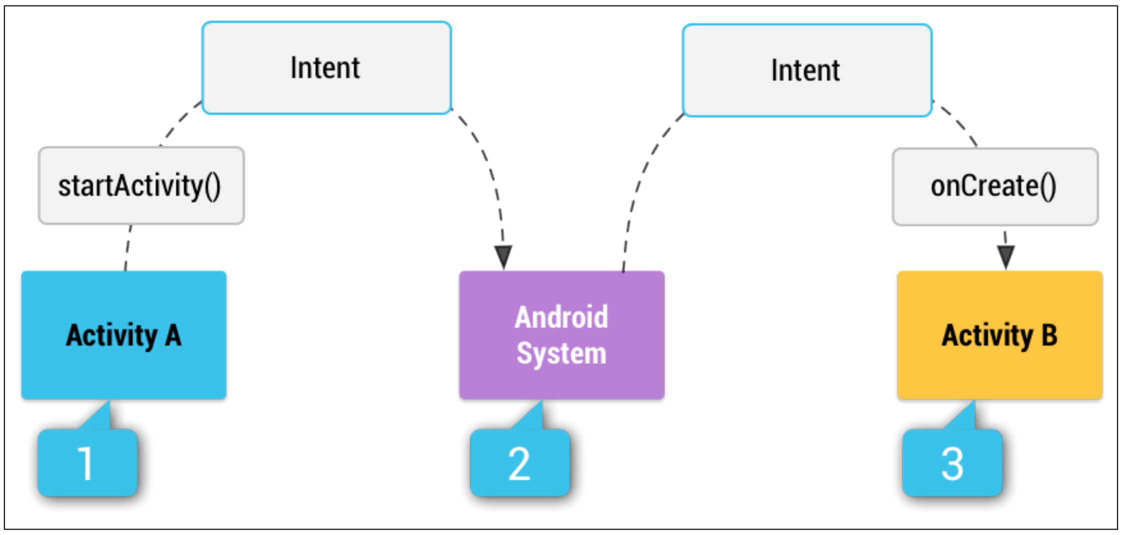


Figure 3.3: Illustration of how an implicit intent is used to start another activity

3.7.3 Services

A Service [78] is an application component that can perform long-running operations in the background and does not provide a user interface. Another application component can start a service and it will continue to run in the background even if the user switches to another application. Additionally, a component can bind to a service to interact with it and even perform interprocess communication (IPC). For example, a service might handle network transactions, play music, perform file I/O, or interact with a content provider while executing in the background. Just like with activities, other apps must know the fully-qualified name of the service, and must have sufficient permissions required to access and interact with it.

3.7.4 Content Providers

Content Providers [70] are used to encapsulate data, and are the standard interface that connects data in one process with code running in another process. A Content Provider represents data as one or more tables similar to a relational database. An application accesses the data from a Content Provider with a Content Resolver object. The Content Resolver is a single global object per application that provides access to Content Providers. The Content Resolver accepts requests and resolves these by directing them to the Content Provider, which acts as an abstraction layer between the repository of data and its external appearance as tables. The Content Resolver stores a mapping from authorities (URIs) to Content Providers, which provides a simple and secure means of accessing other applications' Content Providers in the Android ecosystem. Fig. 3.4[4] shows a Content Provider and its defined interfaces that apps use to interact with the data.

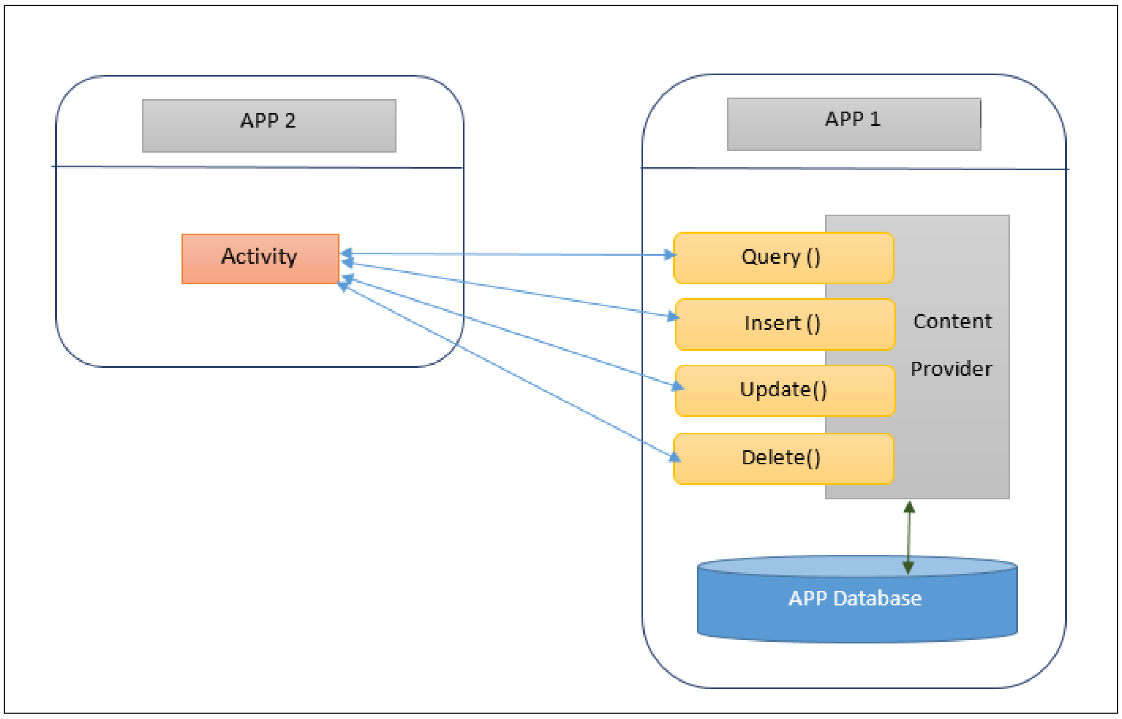


Figure 3.4: A Content Provider and its defined interfaces

The Content Resolver includes the CRUD (create, read, update, delete) methods corresponding to the abstract methods (insert, query, update, delete) in the Content Provider class. The Content Resolver does not know the implementation of the Content Providers it is interacting with as each method is passed as an URI that specifies the Content Provider to interact with. The Content Resolver object is situated in the application's process and the Content Provider object is in the application that owns the provider, and both automatically handle the required inter-process communication needed to send queries and results. A content URI is an argument passed that contains the URI used to identify data in a Content Provider. Content URIs include the symbolic name of the entire provider (its URI authority) and a name that points to a table (a path in the data). The Content Resolver object parses out the URI's authority, and uses it to *resolve* the provider by comparing the authority to a system table of known providers. The Content Provider uses the path part of the content URI to choose the table to access. The URI, the path and its parsing can be defined through code, which allows handling the URI depending on the requirements. Listing 3.4 shows a URI authority corresponding to a Content Provider where `msc.prototype.demo.contentprovider` ↩ is the provider's authority, and `context` is the table's path. The string `content://` is

[4]source: http://www.compiletimeerror.com/2013/12/content-provider-in-android.html

the scheme, and is always present as it identifies the URI as a content URI.

```
content://msc.prototype.demo.contentprovider/contexts
```

Listing 3.4: Example of a Content URI for accessing the *contexts* table within a Content Provider

To retrieve data from a provider, an application needs *read access permission* for the provider, which cannot be requested at run-time. The app needs to specify this permission in its manifest using the `<uses-permission>` element and the exact permission name defined by the provider. The exact name of the read access permission and any other access permissions used by the provider may be available in an app's documentation if the developer has explicitly decided to share it with others. If the Content URI or access permissions are unavailable, the app cannot access the data within the Content Provider. If a provider's application doesn't specify any permissions, then other applications do not have access to the provider's data.[5]

[5] It is possible to access data in a Content Provider without proper access permissions by sending an intent to an application that already has the required permissions and receiving back a result intent containing *URI permissions*. These permissions are limited to a specific content URI that lasts until the activity that receives them is terminated.

Part III

The Contextual Data Sharing Model

4

Context Definition

"Context is worth 80 IQ points."
– Alan Kay

4.1 Introduction

Different apps have access to different kinds of information. Even if this information were to be shared, apps need to understand the information's context in order to utilize the information to present related services. For example, if the *Movie* booking app shared the *Movie* date/time information with other apps, without a proper reference to the context (in this case - a movie) the apps will only see the information as representing time. In order to utilize the information contextually, it is important to refer to context for any data. Along with a reference, the app also needs to access the various fields of information available in relation to the given context so that it can provide services designed around that information. For example, the *Movie* context should always have the *Movie* title, the show time, and the theater *Location* as key pieces of information that make up the context. Other information such as for ticket and seats is optional, and only adds to the contextual information already present about that event. Therefore, in order to utilize contextual information, an app needs to recognize the context and the fields of information available within that context.

Different apps can represent the same context in different ways. It is important to ensure that all apps interpret contextual information in a similar way. This is important for sharing contextual information across apps and to prevent confusion that may arise from ambiguity in the information. For example, some apps may assume that the *Movie* context contains ticket information, while other apps may not. This will create problems when these apps share contextual information that both interpret differently.

The sharing and understanding of contexts in apps that look for fields of information not available or not supported by other apps creates different uses of context that introduce differences in the sharing of contextual information.

The apps use Context Definitions to prevent such problems and enable sharing of contextual information in a uniform way. Using Context Definitions allows the apps to define and share contextual information that is interpreted similarly across all apps that use the definition. This provides a common information format that is structured and understood by all apps using the same definition, which makes sharing of information easier.

4.2 Objectives

These are the three main objectives fulfilled by Context Definitions:

1. To allow apps to recognize the context of shared information;
2. To represent the various fields that are defined or present for a particular context;
3. To define a structure for contextual information that different apps use to represent contextual information in a uniform format.

4.3 Definition

A formal definition of context used for the purpose of this research is based on extending Deys [22] definition: *Context comprises of any information related to or affecting the users activities and tasks. This information includes time, location, weather, sensor information, and all information the user is presented with or enters on or related to a task.*

The information shown by apps to the user and the inputs and choices made by the user also constitute contextual information as they are relevant to the task at hand. This information can be used contextually and therefore also comprises useful context. The app responsible for displaying or accepting such information is tasked with declaring the contextual nature of acquired information.

4.4 Representation

Contextual information can be broadly classified into different types based on their use. The different contexts are represented by a schema based on the information they represent. Each schema is referred to by a unique name and a set of fields that attribute the different information pertaining to the context. Apps use this schema as the definition for that particular context when accessing related contextual information.

Listing 4.1 below shows the schema for the *Event* context :

```
Event {
  Title
  Date/Time
  Location
  Contacts
}
```

Listing 4.1: Schema for *Event* Context

The schema has a unique name called "*Event*" which defines the nature of contextual information contained within it. The title, date/time, *Location* (representing the GPS co-ordinates) and *Contacts* (a Contact object similar to the phonebook) are used to refer to the information represented by an *Event* context. These fields together provide information that tells which Event (title) is taking place where (Location) and when (date/time), and who will be attending (Contacts).

Apps use the definition to instantiate context objects for that particular type of context. When an app uses an *Event* object, it uses the schema or definition to interpret the information available within the *Event* object. Whenever it refers to an *Event* context, it knows that there are title, date/time, *Location* and *Contacts* fields accessible to it. By making this definition of *Event* context uniform across different apps, the representation of an *Event* object and its associated contextual information is also uniform across all apps. This ensures that different apps represent a particular context in the same way irrespective of how it was generated or acquired.

4.5 Extending context

Certain contexts can be modeled as an extension to existing contexts. This allows us to add more information to existing contexts, and also makes representing additional contexts easier. The extended context only has to define the additional fields. The fields from the base or parent context are implicitly included in the schema of the extended context. This creates a relation or compatibility between contexts which reflects how contexts are used in real-world use cases.

For example, a schema for *Movie* context is be described in Listing 4.2. Some of the fields are similar to the schema of *Event* context. The *Movie* Title is similar to the *Event* title, the show date/time is similar to the *Event* date/time, the theater *Location* is similar to *Event's Location* and both schema have a contacts field representing the people that will be attending the said event. The Ticket and Seat Information fields in *Movie* context do not have a corresponding field in *Event* context, and are therefore unique to the *Movie* context. This shows that in terms of contextual information, the *Movie* context is similar to the *Event* context, and shares some of its fields. Contextually we can say that *a Movie is an Event*. Which means that a *Movie* context is a special case of an *Event* context, or that a *Movie* is a type of *Event*. Therefore we model the schema for *Movie* to be an extension of the schema for *Event*. We define the schema for *Movie* as shown in Listing 4.3 using only the fields that are added such as Ticket and Seat information. The other fields belonging to *Event* context are implicitly included in *Movie* context.

```
Movie {
  Movie Title
  Show Date/Time
  Theater Location
  Attending Contacts
  Ticket Information
  Seats
}
```

Listing 4.2: Information in *Movie* Context

```
Movie (extends Event) {
  // fields implicitly included from Event
  Movie Title
  Show Date/Time
  Theater Location
  Attending Contacts

  //fields defined
  Ticket
  Seats
}
```

Listing 4.3: Information in *Movie* Context

An app instantiating a *Movie* context object has access to all fields related to *Movie* context. This includes the fields implicitly inherited from the *Event* context, and the fields declared in the *Movie* schema. For apps, all fields in the *Movie* schema, including those inherited from *Event* schema appear as part of the *Movie* schema. When using the *Movie* context object, the app will store the *Movie* title in the title field, the theater *Location* in the *Location* field, and the show time in Date/Time field.

If we structure the contexts according to how they extend other contexts, we get a tree representing the hierarchy of contexts. The root of this tree is an abstract entity called Context, which acts as a common ancestor. Having a common ancestor and a hierarchy allows for generalization of contexts through which contexts can be reused based on their placement in the hierarchy. For example, if a Calendar app provides services for *Event* context, it will not require modifications and duplication of effort in order to provide the same services for *Movie* context. With generalization, the Calendar app will receive an instance of the *Movie* context as an *Event* context which it can recognize and act on. This allows the Calendar app to provide services for *Event* context as well as all contexts that have extended from *Event* without additional modifications or the need to recognize the new contexts.

Relation between extended contexts

A context hierarchy as depicted in Fig. 4.1 contains the context P with all contexts extending P situated below it. The context C_1 and C_2 extend P, and context C_3 extends C_1. The extending context P is called the parent of the extended context C_1. Also C_1 is called a sub-type of P.

The context C_3 can be generalized to C_1, and C_2 can be generalized to P as they are the children of the respective contexts. Therefore, we can generalize C_3 to P as relations are associative.

In general, all contexts situated below P in the hierarchy can be generalized as instances of P since it is their common ancestor. Therefore, if a service is designed to handle context P, and it receives any context C_x that has been extended from P or any of its descendants, it will receive the context object as an instance of context P. While accessing this context object, it will have access to only those fields defined in the schema for P. Thus, the service works with P and all its descendants C_x in the hierarchy without modifications or special handling. This allows for contexts to be extended freely while still allowing apps that handle a particular context to receive and work with new contexts extended from it.

Single inheritance

To simplify the hierarchy and reduce complexity, a context can extend only one other context. If a context were to extend more than one parent, it could create confusion regarding access to similarly named fields. For example, context A contains a field called Title. Context B and C both extend A and inherit the Title field. Context D extends both context B and context C, and inherits the Title field from both of them. Now when an object of context D is generalized to an instance of context A, it creates ambiguity regarding the field Title. There are two copies of the Title field, inherited from context B and C which can be used to represent the Title field. In this case, it is not clear whether the Title field inherited from B should be used or the one from C. This

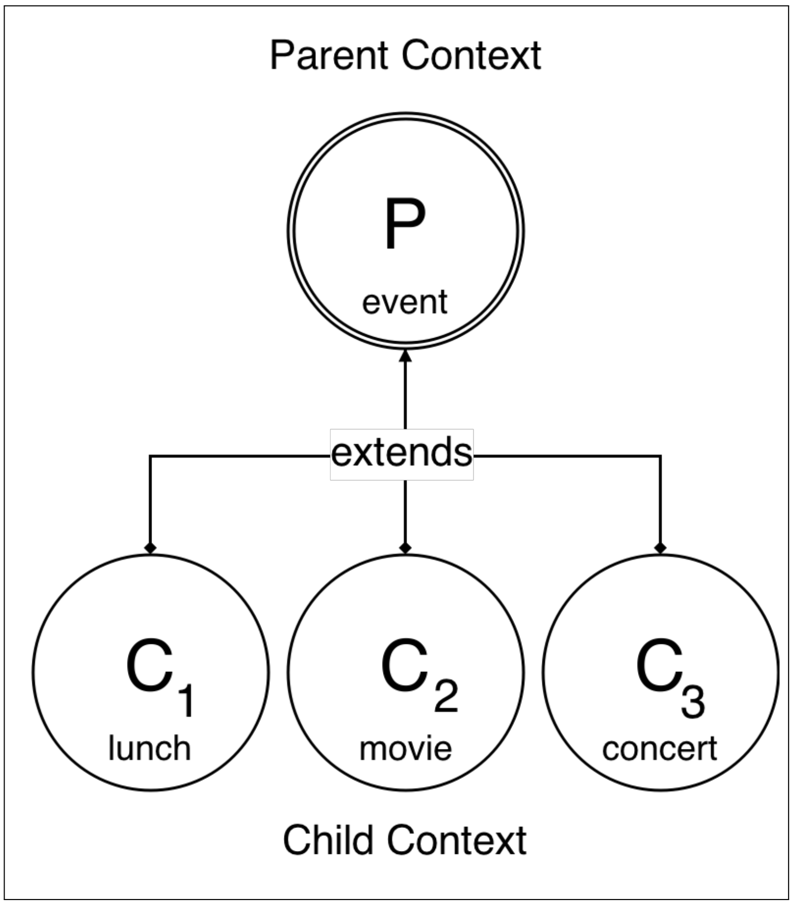


Figure 4.1: Extended contexts in a Context hierarchy

leads to confusion and complexity in design, which can be avoided by forcing a context to extend a single context only. This problem is also known as the diamond inheritance problem which is one of the complications associated with multiple inheritance [79].

4.6 Embedding context

Contexts that represent a part of the contextual information of another context are embedded as sub-contexts. The embedded context acts like an attribute of the parent context while having its own distinct schema and definition. In that case of *Event* context, a *Location* context provides information about where the *Event* is taking place, and is therefore embedded in the schema of the *Event* context. An app referring to the *Location* in an *Event* context object will be aware of *Location* being a separate context and will be able to access all of its fields. The Location field acts as a container that holds all the information about a particular location associated with the Event context. The schema for the *Location* context is given in Listing 4.4. By including the sub-context within the schema of the main context, apps have access to related

```
Location {
  placename
  co-ordinates(lat,lon)
}
```

Listing 4.4: Schema for *Location* Context

contextual information within a single object.

The embedded contexts can be updated without knowledge of the contexts that embed them. For example, an app that provides location-based services such as navigation, that updates the co-ordinates in a *Location* context to denote a particular place. If this information is not linked with the *Event* contexts taking place at that particular location, the contexts will have outdated *Location* co-ordinates. By linking *Location* and *Event* contexts, updates to any instance of *Location* context are reflected across all *Event* instances that use the particular *Location* context. An app can add or update information to a context without being aware of it being embedded in another context. For example, an app can access and use *Location* context objects without being aware of how certain contexts like *Event* embed *Location* in their schema.

If we create a tree based on how contexts are embedded, we get a figure like the one in Fig. 4.2. The contexts at the top are those that do not embed any other context, and are called Simple contexts. Those that embed other contexts are situated below the contexts they embed and are called Complex contexts. Apps that use complex contexts must be aware of all sub-contexts in order to access all the contextual information associated with that context. Using a simple context does not require knowledge of contexts that embed it since the contextual information in a simple context is independent of the contexts that embed it.

Services from apps that handle simple contexts can be utilized by apps that offer services for complex contexts. This allows re-use of functionality and sharing of contextual information in an efficient way. For example, even though the *Event* context embeds *Contact* contexts in its schema, it does not have to provide management of contacts. This service is provided by the Phonebook app which allows users to add and update contact information, which is then automatically reflected in the Contact context in an instance of *Event* context.

4.7 Example Use Case: Movie Ticket Booking

In the *Movie* booking use case if the apps use Context Definitions to identify and share contextual information, the resulting user experience becomes simpler and better as the apps use the available contextual information to offer related services. The effort required to access and utilize the contextual information becomes significantly less

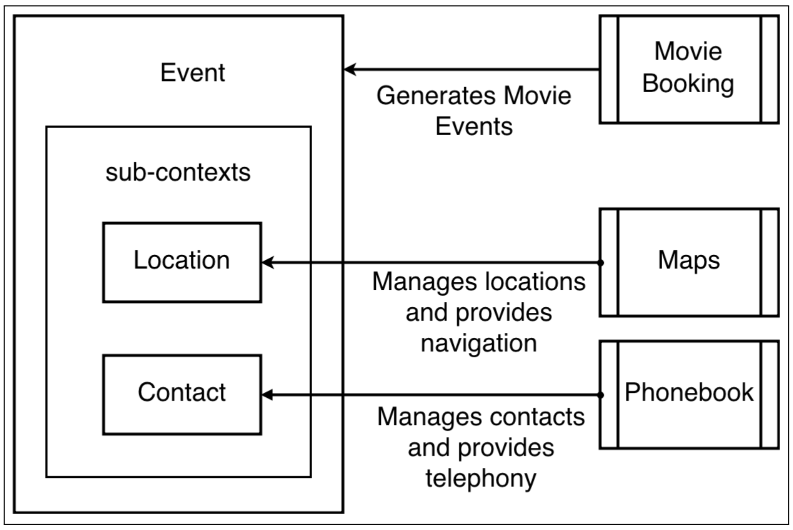

Event
sub-contexts
Location
Contact
Generates Movie Events
Manages locations and provides navigation
Manages contacts and provides telephony
Movie Booking
Maps
Phonebook

Figure 4.2: Different apps managing sub-contexts within *Event* context

compared to the apps trying to acquire this information on their own. By extending and embedding contexts, the user can use more contextual services with less effort for app developers to provide such services.

The Calendar app that provides notifications for *Event* contexts will also provide notifications for *Movie* contexts since it is extended from Event. This enables the user to use Calendar services like notifications and reminders for *Movies*. The *Location* services provided by a Map application are based on using the *Location* context. The *Event* context embeds a *Location* context in its schema, and which is implicitly included in *Movie*. By using this *Location* field, the Map application can provide directions to the theater without requiring the user to input this information. This saves effort and duplication of information, and provides easier navigation for the user. *Contacts* are handled by the Phonebook app that offers users the ability to change contact information. A change or update to a particular contact is reflected across all contexts that embed it. If a particular contact was added to the *Movie* context, and the contact information was changed by the user through the Phonebook, the contact in the *Movie* context will also reflect this updated information. This allows for up-to-date information to be available across all shared contexts and allows apps to share and update information freely. The *Movie* booking app only has to create the *Movie* context related to the ticket booking. Other services such as notifications and navigation are provided by other apps without being aware of how the *Movie* context was created or updated. This allows for a variety of related services to work together on shared contextual information without being aware of how other service use the same or related contexts.

4.8 Responsibility of owning Context Definitions

While definitions form an important aspect of recognizing, declaring and using contexts, it is important to note who manages the creation and handling of definitions. While we provide no recommendation as to who should create definitions, it is important to note from an implementation point of view that the definitions have to be available at the system level as well as to app developers to include in their apps. Therefore some level of involvement from OS manufacturers may be necessary in order to get these definitions on devices and in APIs related to the app development.

4.9 Summary

A Context Definition is used to represent contexts uniformly across apps and devices. A definition constitutes a schema with a unique name and fields that make up the contextual information associated with that particular type of context. Apps use the context definition to initialize context objects of that context type. A context can extend other contexts, which implicitly includes all fields from the parent context in its schema. While the context definition only contains the additional fields defined, apps can also access the implicitly included fields in the context object. A context can embed other contexts, where the embedded context is used as a field representing contextual information about the main context. Extending and embedding contexts allows reuse of functionality, information and services. This makes it easy to develop apps for contexts as services can reuse other services that handle a context without dependence on how the other contexts extend or embed it.

5

Context Database

"Knowledge is of two kinds. We know a subject ourselves, or we know where we can find information on it."
– Samuel Johnson

5.1 Introduction

Context Definitions allow an app to structure contextual information through a schema which other apps can interpret and understand. After an app has identified or created contextual information, it needs to store the context so that it can retrieve and/or act on it later. The Context Database allows apps to persist contextual information between uses. The Context Database acts like a central repository, where apps can save and retrieve contexts. The app that generates the context inserts it to the Context Database and other apps query the Context Database to retrieve this information at later times. The saving and retrieval of contextual information is asynchronous, which allows apps to access contextual information when they need it.

5.2 Overview

The Context Database stores contextual information using Context Definitions, which allows for easier sharing of contextual information between smartphone applications. The Context Database assumes responsibility for management and storage of the contexts. It will store contexts generated by an app even after the app has been uninstalled. This preserves the contextual information on a device which would have been otherwise lost. Apps can add or modify contexts in the Context Database and can share them with other apps indirectly through the Context Database.

5.3 Design and Structure

The design and structure of the Context Database is based on the type of database or data store used in the implementation. The features and services supported or provided by the database software are be used to determine the types of queries the Context Database supports. In general, the database software should allow CRUD (Create, Request, Update, Delete) operations for a basic working of the Contextual Data Sharing Model. This includes a way to add contexts to the database, to query contexts based on their type, update context entries in the database and a way to delete contexts.

The Context Database should be managed by the system so as to maintain independence from any particular app. On most implementation platforms, this also includes implementing the Context Database outside the memory space of any user app. This safeguards the Context Database against any malicious processes that may crash the database or corrupt data. If the smartphone operating system uses Sandboxing to isolate process data, it can be used for additional security measures.

The location of the Context Database does not have an impact on the contextual data sharing model, but can potentially have some impact on the user experience. The actual storage backend depends on the implementation and can be based in the cloud, or in a local data store.

5.4 Deleting Contexts

Apps are prohibited from deleting contexts in the Context Database due to security concerns. The deletion of contexts in the Context Database is handled by the system. This is done to prevent loss of information and to prevent misuse of the deletion operation by apps. If apps are allowed to delete contexts from the Context Database, it can lead to complications and loss of information. For example, if some app no longer requires a context and deletes it, it can lead to loss of information for another app that may still be using the same context. Also deletion causes permanent loss of information which could have been useful in the future. In cases where there is a link or a relation between contexts such as embedding, the deletion can cause errors. If a context being deleted is referenced as a sub-context in another context, it will lead to loss of contextual information. This can affect the operations when querying for the context next time, as the query may return a reference to a non-existing context which will cause errors, and may stall database operations or crash the app.

5.4.1 Factors affecting efficiency

The size of the Context Database will have an impact on performance of queries. A database with more records (having greater size) will take more time to execute the same query than a database with fewer records. To keep the database within an acceptable range of values, the Context Database manages the deletion of contexts through a Deletion Policy. The Deletion Policy is used to manage the size of the database and to keep the average query execution time under permissible values. The time taken by a query to successfully execute depends on the number of records, device configuration and app usage. The device configuration is fixed, and cannot be changed. A high specification device can execute complex queries faster than a comparably lower specification device. Therefore the effect of device configuration on query time will be fixed. The app usage depends on user interactions, which can vary greatly and are difficult to classify into expected user interaction patterns. Multiple apps accessing the Context Database simultaneously will also have an impact on its performance. Therefore, the only factor which is manageable and can be used to keep the Context Database operating efficiently is managing the size of the database.

5.4.2 Deletion Policy

The Deletion Policy is activated whenever the size of the database or the number of contexts reaches some *threshold* value t that is determined as the maximum size of the database for which the average query time is within an acceptable range. The value of t will vary between devices depending on the specification, use of apps and available space.

The Deletion Policy is an implementation consideration addressing the following points:

1. *What - Selecting contexts to delete*: The Deletion Policy selects the contexts to be deleted based on some parameter like time, size, cost or need which marks the contexts to be deleted.

2. *When - Condition to trigger deletion*: Along with the value of t, it is important to consider other activities that will affect when the deletion can be safely run. If some apps are querying contexts in the database, then the deletion policy should be stalled until such apps have finished querying. Also, the phone state, active processes and performance and battery life considerations should also be used to come up with an optimum time for initiating the deletion.

3. *How - Handling context relationships and conflicts*: If a context entry about to be deleted has a sub-context, should that be deleted as well? Or if the context

is a sub-context of some other context, what should be done about the reference. Such considerations can be affected by the implementation platform and backend software used to implement the Context Database. In general, the following cases will help when implementing the Deletion policy:

- When the context to be deleted has a sub-context;
- When the context to be deleted itself is a sub-context;
- When the context to be deleted has some complex relationship with another context.

5.5 Performance Considerations

The apps using the contextual data sharing model will be interacting with the Context Database whenever they need contextual interactions. In some use cases, this interaction can happen on a regular basis. Multiple apps can try to access the contextual information in the Context Database simultaneously. The user experience in many apps is dependent on the use of contextual information. For example, in Calendar the Events are shown on screen in some format. For this, the app must query the Context Database to retrieve all the Events. Simultaneously, another app may also be requesting Events from the Context Database, to check if there is an upcoming Event for the day. Various other apps may also query the Context Database for different purposes. If the Context Database does not respond in a specified time period, it can affect the UI interactions of the app. If there is a delay in returning the Events to Calendar, then the user may notice a time gap where nothing is happening on screen, or things are being updated slowly. To prevent such delays and to offer a smooth user experience across apps, it is necessary that the Context Database has to be efficient in its queries, returning the requested results to apps in a shorter time period, which allows for faster UI interactions. To make the efficient use of available resources, the database design and schema should be kept simple so as to not impact the system performance.

5.6 Summary

The Context Database is used to persist contexts through a backend consisting of a database or data store software. Apps query the Context Database to insert and retrieve contexts. The sharing of contexts is done asynchronously as apps can query for contexts without being aware of which app performs the inserts. The software used to implement the Context Database is an implementation choice as long as it follows the basic CRUD operations required for the functioning of the contextual data sharing model. A few performance considerations such as access time and average query execution time need to be taken into account to keep the user interactions within apps running smoothly. The managing of query execution time is tackled by managing the size of the database. The Deletion Policy is responsible for handling the deletion and is an implementation choice as the actual deletion operations are affected by the design, schema and software being used.

6

Contextual Data Sharing Model

“You never change things by fighting the existing reality. To change something, build a new model that makes the existing model obsolete.”
– Buckminster Fuller

6.1 Introduction

The Contextual Data Sharing Model describes the various components required to create a framework that enables applications to store and share contextual information. The responsibility of each component and its relation with other components is specified in the model. Designing and implementing the components individually or separate from other components can create a disparity in the working and sharing of information. The Contextual Data Sharing Model describes the responsibilities of various components, which allows a design-level understanding of the working and limitations of each component. This allows the components to be designed and implemented in different ways as long as each component fulfills the responsibilities in the model. This allows the model to be designed and implemented on a variety of platforms.

Components in the model constitute the Context Definitions, which allow applications to identify contextual information, and to share it with other applications. The Context Database provides a way to persist contexts, that allows sharing of contextual information across apps. Utilizing Context Definitions and storing it in Context Database allows collection of information related to a context in a single, uniform and identifiable structure, and allows it to be stored and queried by apps independently. The module responsible for managing interactions between apps and the Context Database is the Context Manager.

The Contextual Data Sharing Model ties in the various components to provide contextual information to apps. The availability of such contextual information allows apps to design services that allow users to carry over tasks seamlessly across different apps.

6.2 System Model

The various components of the Contextual Data Sharing Model are depicted in Fig. 6.1. The components are separated into *System* and *User App* partitions based on their responsibility, access and usage. The Context Database and Context Definitions lie within the System partition. The Context Manager lies within the User App partition. The partitioning of components allows separation of responsibility and provides security and stabilization to the Contextual Data Sharing Model.

6.2.1 System Components

The components in the system process are managed by the system and lie outside and separate from any user app's process. The separation of Context Database allows control of access to information, and provides stability and security. It acts as a barrier to prevent corruption of data and malicious access from harmful apps. The separation also allows changes to the underlying components such as the Context Database without affecting the overall design of the Contextual Data Sharing Model. The changes made would be transparent to the user apps, who would continue to interact with the Context Database in a specified manner.

Context Definitions

The Context Definitions are used by the system to instantiate context objects. Whenever an app creates a context object, the system refers to the Context Definitions to create an object of the specific context. Since the Context Definitions represent the structure of shared contexts, they are immutable and managed by the system. This prevents any app from corrupting the definitions stored in the system.

The Context Definitions can be used to instantiate context objects outside of the Contextual Data Sharing Model. Developers can use the context objects in their app without the information being added to the Context Database. This allows the Context Definitions to be used freely in any way the developers see fit.

Context Database

The Context Database acts as a repository where apps can store and retrieve contexts. This allows apps to share contexts without a direct interaction between them. The

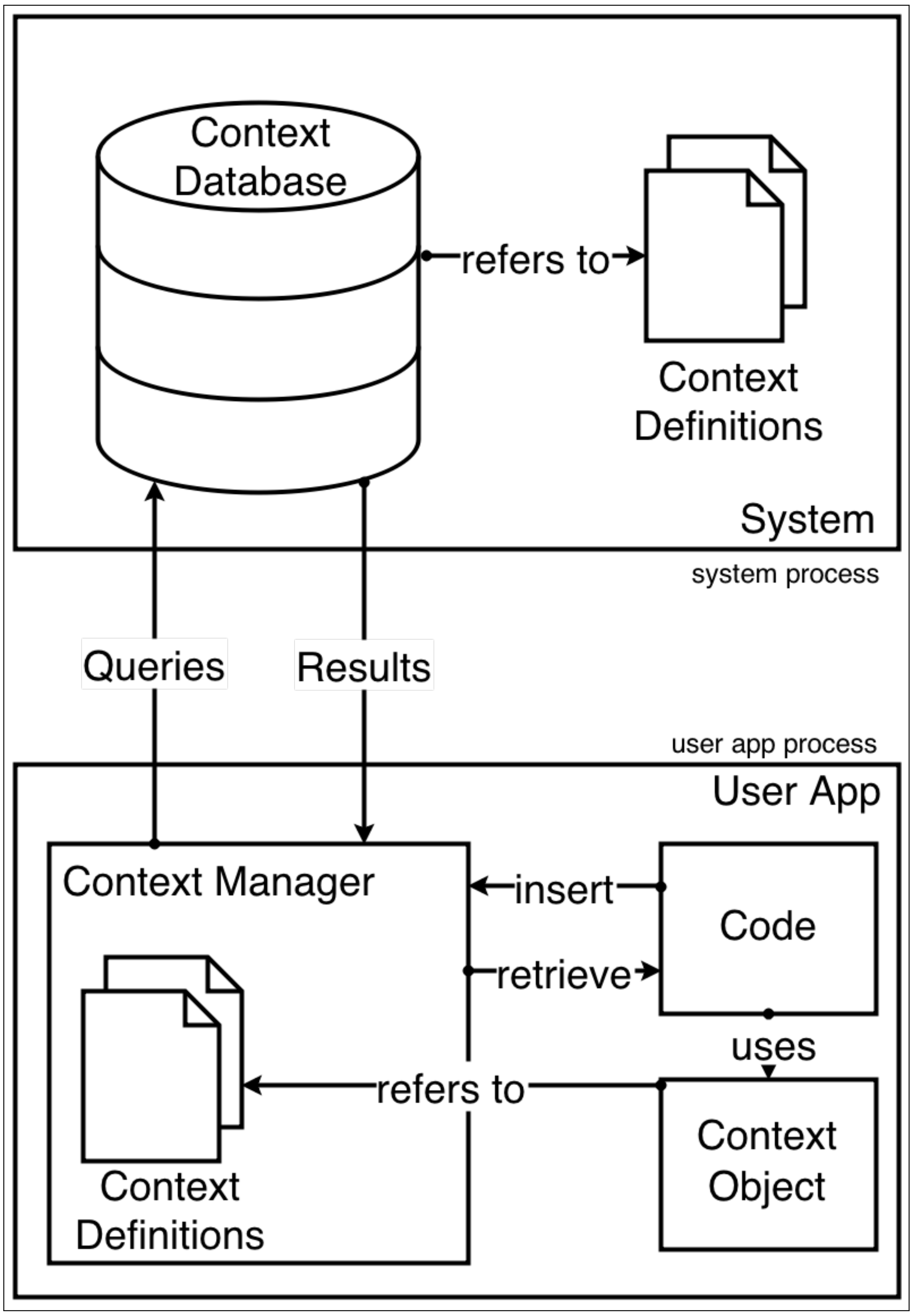


Figure 6.1: Overview of the Contextual Data Sharing Model

Context Database comprises of the actual backend data store or database used to store the contexts, and the API used to interact with it. All access to the Context Database is through the API, which allows a limited selection of queries to be executed by the apps. The selection of queries is based on the functionality of services the Context Database offers. The actual API and queries are dependent on the implementation of the Context Database.

6.2.2 User App Components

Context Manager

The Context Manager acts as the middleware between an app and the Context Database. Apps that wish to interact with the Context Database can only do so through the Context Manager. This forms the access control mechanism used to restrict access to the Context Database. The Context Manager performs the necessary communications to request queries to be run in the Context Database. It is also responsible for interpreting the response from the Context Database in a format understood by the app. This involves requesting context entries from Context Database and instantiating context objects from them. The Context Manager is also responsible for performing any error checks and verifications on the correctness of a context being added to the database. More information about the Context Manager is given in Section. 6.3

Using Context Objects

An app has no ownership or control over the context object after it has been inserted in the Context Database. Other apps can update or modify contexts irrespective of whether the context was inserted by them. This allows various apps to access and update contextual information, which results in greater sharing of information keeps the information updated. A context updated by an app is instantly reflected across all other contexts that use or refer to the changed context. This allows any app to update contextual information available to it, and allows other apps to receive the updated contextual information without being aware of the update.

6.3 Context Manager

The Context Manager is a user app component, part of the user app itself. The Context Manager is embedded in the user app as a separate module or library, which cannot be changed or modified by the developers or the app itself. Each app will hold its own instance of the Context Manager at runtime or during execution. This creates a greater abstraction between the user apps and the system components, which permits more freedom in implementing the various components of the Contextual Data Sharing Model.

The Context Manager provides a way for apps to insert and retrieve contexts from the database without interacting with the Context Database directly. Additional features such as error checking for context objects are based on the implementation of the model and are not necessary for the working of the Contextual Data Sharing Model.

6.3.1 Errors generated as part of the user apps' process

Since the Context Manager is a part of the user app, all operations performed by the Context Manager are executed as part of the user app's process. If any fault of error is generated during an operation in the Context Manager, then the fault or error is generated as part of the user app. This keeps all faults and errors generated within the app from affecting other components, and prevents apps from crashing the Context Database. If the app tries to run some operation within the Context Manager that causes instability in the system or potentially crashes any process, then the app's process itself will become unstable and will crash. The unstable operations run as part of the Context Manager are actually being run in the user apps process. This provides stability and security to other apps and the system from potential crashes and errors. This also allows the Context Database to continue working in case a query causes errors, as the error is generated in the process containing the Context Manager. This allows other apps to interact with the Context Database without interruptions, and allows the Context Database to continue to handle requests from other apps.

6.3.2 Checking queries before execution

The Context Manager acts as a mediator between the app and the Context Database. It accepts requests from the app, creates appropriate queries and sends them to the Context Database. It interprets the results of those queries and creates context objects which the app understands. For the app, the Context Manager is the sole interface to the Context Database and all inserts and requests are completed through it. This allows all operations requested by the app to be checked for errors or completeness before passing on to the Context Database.

For example, if an app requests an *Event* context to be added to the database, the Context Manager will first check whether all fields of the context object are properly initialized. Only then will it send the query to the Context Database to add the context object. Doing so provides additional security as incorrect requests are not sent to the Context Database but are handled by the Context Manager itself. This reduces the checks performed by the Context Database, and allows faster execution of requests.

This increases the efficiency of operations run by the Context Database and allows more apps to interact with the Context Database simultaneously.

The Context Manager requests queries to be run on the Context Database on behalf of the app. The queries requested by Context Manager are based on a fixed subset of queries explicitly allowed to be run by user apps, and cannot be changed. If a malicious app somehow manages to make the Context Manager request execution of queries other than those specified, the request will fail as the Context Database will not execute the query. So operations and requests from the Context Manager to the Context Database are safe, and are checked before execution.

6.4 Security Considerations

The contextual nature of information stored within the Context Database can lead to security concerns. These include handling privacy of information and preventing corruption of data. It is important to be aware that sensitive information may be stored within the Context Database and it is important to ensure that proper checks are put in place to access such information.

One way many smartphone operating systems (including Android and iOS) handle access to information and resources is through the use of Permissions. The app requests or declares the intention of using certain information or resource and requests access to it. The system or the user, depending on the implementation of such a model, will grant or reject access to the app if they choose to do so. This system of using permissions to explicitly declare the resources and information an app needs to work can also be used to control access to information stored in the Context Database.

Permissions can be utilized to restrict access to the Context Database and to the use of contexts itself. An app can provide explicit read-write permissions for each kind of context they intend to use at the time of installation of during each use of that context. The system or the user can use these permissions to be aware of the information the app is accessing and can act if the information being requested does not match the activities of the app. Separate read and write permissions can be used for apps that want to use contexts from the Context Database, but will not insert contexts. This will prevent apps from corrupting information in the Context Database under the guise of providing some service that relates to only reading of information in the database. A malicious app can still take advantage of the system by offering some service that requires the write permission, and then inserting corrupt data to the Context Database. Such apps can be better scrutinized for suspicious actions through explicit context permissions that declare what contexts the app wishes to read or write from the database. For example, a movie ticket booking app requesting access to *Movie* contexts can be considered as normal. But if the same app requests access to all *Event* contexts, then the app should be scrutinized more carefully.

6.5 Example: Apps using the Contextual Data Sharing Model

In the movie ticket booking use case, the task requires the use of several apps that store the acquired information in a way that is not shared with other apps. This leads to other apps depending on the user to acquire the contextual information, which leads to duplication of information and efforts required to enter it in to the app. With the Contextual Data Sharing Model, each app designs its services on the assumption that contextual information is available to it. In cases where suitable contextual information is not available, the app has the option of asking the user to enter the required information.

The app responsible for generating or interpreting the contextual information stores it in the Context Database so that it can be shared with other apps. The movie ticket booking app saves the appropriate information in the form of a *Movie* context that contains the details about the movie such as title, show times, ticket and seat information and also the theater location and contacts who are attending the movie. Some of this information such as theater location and contacts may not be available at this stage, but can be filled in by other apps who have access to this information.Once the booking app has inserted the contextual information in the Context Database, other apps can access it and provide services based on the information.

The user is likely to perform actions related to the context, which in this case is to forward the details of the booking to other attendees, or to find a route on a map to the theater, or to access seat information once at the theater. Each app that provides one or more of these services access the contextual information of the movie context in the database to provide services that are directly related to the user's task. This is more useful for the user as related services are offered without the user having to navigate within the app or having to enter information.

The Messaging app which is used to send movie details of the booking can utilize the Contextual Data Sharing Model to access the contextual information in to the *Movie* context. It can then offer to insert this information in to the message saving the user the effort of entering the information or copying it from another source. Additionally, it can allow the user the choice of selecting the fields from the *Movie* context to be inserted into the message body, and to use the *Contacts* from the context as the recipients. To make it easier for the user to retrieve contextual information, the messaging app can show the recently used or recently added contexts. Such filtering of contexts based on time is dependent on the Context Database implementation.

By providing information to the user in an easily usable format, the user is more likely to complete tasks in fewer steps. Additionally, the user is presented with information and services most likely required, which increases the quality of user experience and decreases the time taken to complete tasks. When the user opens a Maps application that provides navigational services, the app can retrieve a list of upcoming *Events* from the Context Database to offer a choice of locations for routes. If the user selects a route displayed within the list, the *Location* field of the corresponding *Event* is used as the destination in the navigation. This saves the user the effort of typing in the address of the *Event* and also prevents ambiguity or confusion with similar addresses and areas. Further, the user can select routes for the theater as well due to the model generalizing the *Movie* as an instance of *Event.* This reduces the effort required by the Maps developer to integrate different context types into their app.

Once at the theater, the user would want the ticket and seat information at various times. An app providing location-based reminders can show a notification containing this information for easier access to the user. It can use the *Location* field from the *Movie* context to trigger the reminder, and then show the ticket and seat information from the same context. This saves the user the effort of creating the reminder themselves by entering all the associated information.

By using specific queries that retrieve data based on some parameters, the apps can utilize contexts more efficiently. This allows apps to design more services based on the specific nature of contexts that they can request from the Context Database.

For example, a daily planner app that generates an agenda for the day will query the Context Database for *Events* and other related contexts that occur on a given date. This allows the app to pull in only those contexts which are useful to it at the given moment. Another example can be a restaurant app that queries the Context Database for *Events* and their location to provide recommendations in nearby areas. By filtering contexts, apps can tailor specific services based on the results of such queries.

6.6 Summary

The Contextual Data Sharing Model consists of three parts - the Context Definitions that provide a uniform representation of contexts, the Context Database that stores the contexts and the Context Manager that acts as a middleware between the app and the Context Database. Apps that wish to use the Contextual Data Sharing Model and interact with the Context Database can only do so through the Context Manager. All means of sharing contextual information between the apps is indirect and through the Context Database, whose purpose is to store and allow sharing of contextual information across applications.

The Contextual Data Sharing Model enables the collection of information related to a context in a single structure, and enables apps to provide services using contextual information available in the Context Database. This allows users to carry over tasks across different apps by sharing contextual information related to their actions.

Part IV

Contextual Data Sharing in Android

7

Implementation

"Talk is cheap. Show me the code."
– Linus Torvalds

The implementation described in this thesis is a proof-of-concept meant to test and demonstrate the working and impact of the Contextual Data Sharing Model. The model was implemented on the Android platform, chosen for its openness and ease of modification. The various components were developed with technologies that are native or work well with Android.

The Context Definitions are modeled using Java Classes which allows context objects to be instantiated as Java objects. The Context Database uses SQLite for the backend to store contexts objects. The Context Manager is realized as a static Java class embedded in every application. Components of the model such as the Context Database and the Context Definitions are designed to facilitate porting to other platforms. Other components such as the Context Manager and the implementation of Context Definitions utilize technologies specific to the Android ecosystem and require additional effort for porting to other platforms such as iOS. The code for the implementation is hosted in a git repository at `https://github.com/coolharsh55/ContentProvider`.

7.1 Choice of Platform and Software

7.1.1 Platform

Of the major smartphone operating systems in use today (iOS and Android), Android offers more choice and freedom of implementation due to its openness and availability of source code. The platform allows a greater degree of data sharing than iOS, which allows for easier demonstration of the Contextual Data Sharing Model.

The platform used in the implementation is Android version 4.4.4, also known as *KitKat*. At the time of inception of this research, this was the latest stable version of Android available. A newer version of Android, version 5.0 also known as *Lollipop* was released in November 2014. The newer version does not include any major change that affects the design or the working of the Contextual Data Sharing Model. As such, the existing implementation will work with minor adjustments on the newer platform.

7.1.2 Context Definitions

Contexts can be referenced and implemented in a number of ways. The main concern of implementing Context Definitions is storing the entire contextual information in a single context object. Traditional representations of context such as RDF and OWL use a structured schema to represent contextual information and relationships. Using such formats on a smartphone has the drawback of requiring parsing every time the information is to be used. Furthermore, such formats cannot encapsulate the associated functionality that is associated with a context object. This introduces additional complexity in the design of the system and the code.

Data on smartphone operating systems is often serialized in formats native to the platform and code for easier management and access at a later time. The formats discussed previously are not native on smartphone platforms and will need to be converted or interpreted to native forms before use. This makes the serialization and representation of contexts difficult and introduces additional steps in the code.

Therefore, a context is represented using Java Classes which are native to the Android platform and are familiar to the developers of apps on Android. Using Java to represent the contexts simplifies the code as there is no need for parsing or extraction. All related functionality such as error-checks and marshaling is encapsulated into the class itself, which leads to cleaner code and better management of contexts.

Since Android uses a native Java runtime environment, the Java classes used to represent contexts are instantiated as Java objects. This results in better management of memory during execution and allows sharing of information across apps without conversion. The developers of apps can use data structures and code management techniques related to Java for managing contexts.

7.1.3 Context Database

Local versus Cloud

The design of the Contextual Data Sharing Model allows the Context Database to be implemented abstractly with different backends. The Context Database can be implemented as a cloud-based data store which performs operations in the cloud and returns the result back to the device. Such an implementation would have the advantage of additional features possible due to the increased storage and processing power of the cloud. The drawbacks of such an implementation would include the need to access the cloud provider on every request, the constant dependence on network connectivity for the Contextual Data Sharing Model to work and the effect of network QoS on the time required for various operations.

The contextual information discussed in this thesis is generated and consumed by the apps installed on a smartphone. A local database situated on the device and working as the backend for the Context Database offers the advantage of keeping all information generated, stored and consumed within the same ecosystem. It also makes it possible to use cloud offloading to offer additional functionality and resources. The database system used in this implementation is situated locally on the device.

Backend database software

It is necessary to use database systems that are stable and readily available as they provide high availability across a large variety of devices. Storing hierarchical data like the context hierarchy in relational database becomes increasingly complex and restrictive as more nodes are added to the hierarchy [80]. The increase in complexity creates difficulty in storing contexts efficiently in the database.

The structure and querying of contextual information is highly dependent on the type of database software used. A RDBMS restricts the flexibility to change or update the context schema without first migrating all data outside the database. In contrast to this, NoSQL database solutions [81] such as graph and document databases are more suitable for storing hierarchical data. A graph database is useful for storing relationships between various contexts, which can increase the quality of contextual information stored on a device. However, the use of graph databases is restricted by the lack of availability of popular and stable products that work well on mobile devices.[1]

The Context Database component in the specified implementation uses SQLite version 3.7.11 [82], which is a popular serverless SQL database available on all major platforms. SQLite is pre-installed on all Android devices, and the version specified is found on all devices running Android 4.4.4 (Kit-Kat).

[1]Available NoSQL graph database for Android include Titan (`https://wuman.github.io/titan-android/`), Neo4j (`https://github.com/neo4j-contrib/neo4j-mobile-android`) and Sparksee (`http://www.sparsity-technologies.com/`), all of which are under development and do not have the level of stability suitable for mass adoption.

7.2 System Model for Implementation

Fig. 7.1 shows the system model for the implementation of Contextual Data Sharing Model on Android. The system components are managed by the Android system and run under the system process. The components in the user app are run under the user app's process.

The Context Definitions are represented using Java Classes, which are pre-installed to provided a uniform implementation across devices. Developers access these classes as part of an API when writing the app code. During execution, apps use the Context Definitions which are a part of the system to instantiate context objects.

The Context Database is managed by the system and is exposed to the apps through an API handled by Android's Content Provider mechanism. For storing data, the Context Database uses SQLite which is pre-installed on Android. The SQLite database used is considered pre-installed and part of the system, though in practice an app was used to instantiate the Content Provider and the corresponding tables.

All queries from an app to the Context Database are handled through the Context Manager. The Context Manager is instantiated as a static Java class in the user app's process, and is responsible for querying the Context Database through the Content Provider interface. It interprets the results returned from the Context Database to instantiate context objects which are then passed back to the app.

Every app has its own instance of the Context Manager which acts as a middleware between the app and the Context Database. Apps use the API provided by the Context Manager to insert or retrieve context entries from the Context Database. The Context Manager in turn uses the API provided by the Content Provider to communicate with the Context Database. Except for the actual database queries, all other operations such as field-checking, marshaling and error-checks are performed by Context Manager in the user apps process. Apps are free to create and utilize context objects. The Context Manager performs the error and instantiation checks only when objects are being inserted or updated in the database.

7.3 Context Definitions using Java Classes

7.3.1 Context Java Class

Context Definitions are represented through Java classes which are then instantiated into Java Objects. Each context type is represented using its own unique Java class. Listing 7.1 represents the *Event* context using a Java class called *EventData*. Each context class is suffixed with *-Data* to prevent clashes with other classes that may have the same name. For example, Android has several classes named *Event* in different

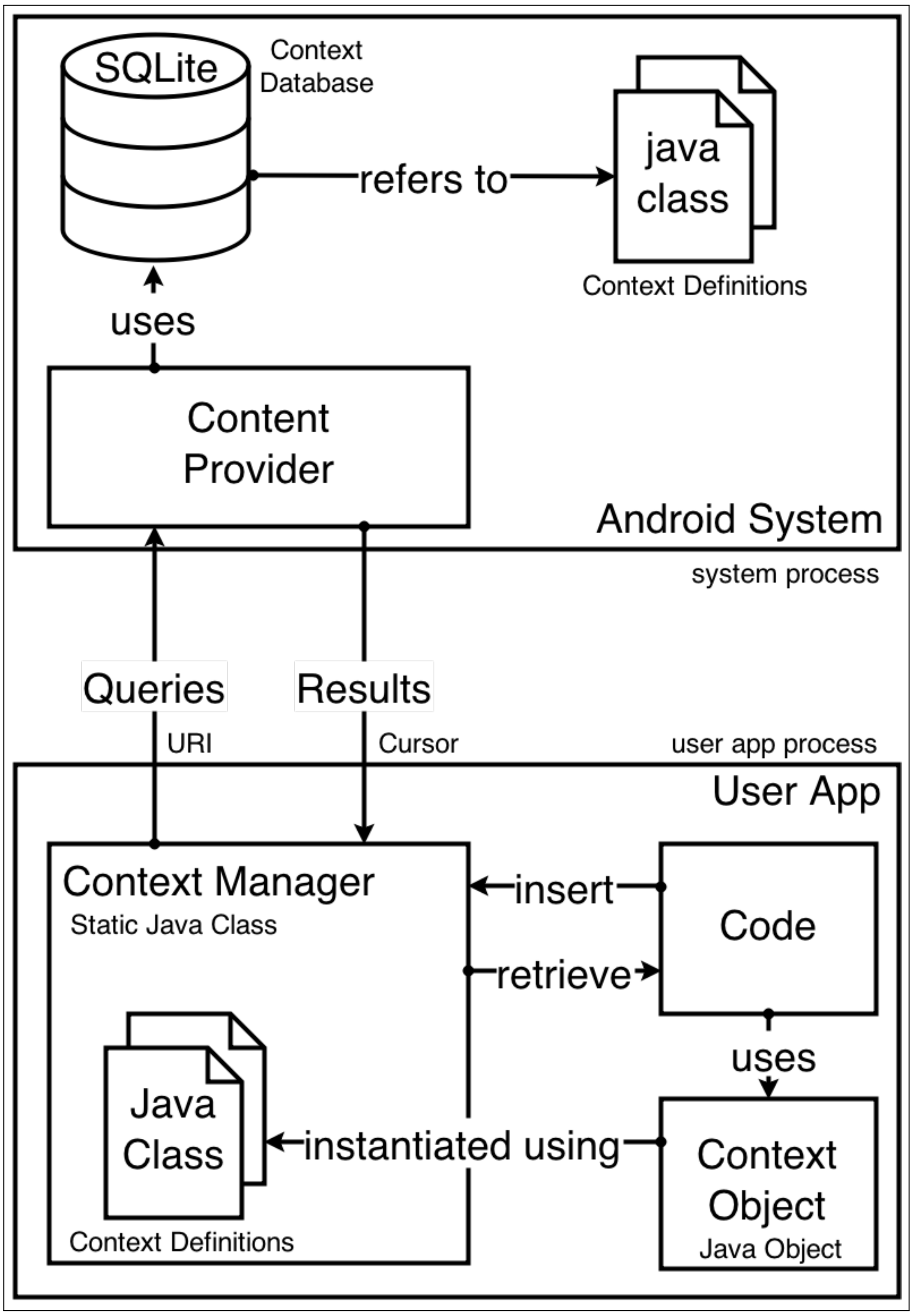


Figure 7.1: Implementation of the Contextual Data Sharing Model on Android

```java
package msc.prototype.context;

import android.net.Uri;
import Java.util.Date;
import android.os.Parcelable;

class EventData extends ContextData {
  String title;
  Date date;
  LocationData location;
  ArrayList<ContactData> contacts;
  Uri uri;
  // other methods and parameters
}
```

Listing 7.1: Schema for *Event* Context

packages. For the sake of brevity, this text omits the suffix when speaking about specific Java classes. For example, the *Event* Java class actually refers to the *EventData* Java class in code. The code listings provided contain the actual class name with the suffix.

All context classes are defined in the package `msc.prototype.context` and are extended from a common context class called *Context*. The *Event* class shown in Listing 7.1 belongs to this package that groups it together with other context classes. The title field is a String that depicts the Event title. The date is a Date object that stores the date and time. The *Location* context is a sub-context that refers to the location of the event. The contacts are stored as an ArrayList of *Contact* contexts.

URI field

The URI field stores links for related information about a context. A URI in Android can be used to refer to information in websites and also in apps. For *Event*, the link can point to the event website or an app that holds this information. Apps that want to handle a particular URI scheme need to register the scheme with the system. This is done by declaring the scheme in the app's manifest. The manifest of an app acts as a declaration of the requirements and components and is used by the system during installation of the app. The system reads the scheme information declared in the manifest during installation of the app and associates that particular scheme with the app. For example, an app registers the scheme *http://* which is used to browse webpages. Anytime the user clicks on a link that starts with the specified scheme *http://* the system will show a list of all apps that handle the particular link. In case of the *Event's* URI field, the link is opened by the system with appropriate apps that can handle the scheme.

Apps can store links to information or services offered by them through this URI

```java
package msc.prototype.context;

import android.os.Parcelable;

class ContextData implements Parcelable {
  private long _id;
  public final long get_id(SystemToken token) {
    return _id;
  }
  public final void set_id(SystemToken token, long _id) {
    this._id = _id;
  }
  public abstract String shortDescribe(); // short description
  public abstract String longDescribe(); // long description
  public abstract boolean checkFields(); // perform field checking
}
```

Listing 7.2: Schema for *Event* Context

field. A restaurant booking app can store a link that points to the booking activity within that app. Clicking on this link will take the user directly to the activity in the booking app where they can manage the booking or use additional services offered by the app.

The URI links can be opened or triggered inside any app as the system handles all links and opens the appropriate app. This allows apps to display links to information and services belonging to other apps. This allows related services to be displayed which helps users to perform tasks related to the context without explicitly switching apps.

7.3.2 Abstract Context class

The abstract *Context* which acts as the ancestor of all contexts is realized using the *ContextData* Java class. Listing 7.2 shows the *Context* class and its associated fields and methods. It contains an *id* field of type long (long integer, holds larger values) that represents the particular *id* of that context object in the Context Database. To retrieve or set this *id*, the get/set methods require an object of type *SystemToken*, which can only be generated by the system. This ensures that only the system has access to the *id* field and that apps cannot change or modify it. By declaring the *id* field in the *Context* class, all contexts that extend it inherit the *id* field as a property of that class.

The abstract methods *shortDescribe* and *longDescribe* are used to retrieve information about the context in a String format. This can be used by apps to display information about contexts in an easier way than manually accessing the various fields. For example, the *shortDescribe* method of the *Event* class returns information about the title and date/time of the event. The *longDescribe* method returns more information including the title, date/time, location, contacts and a URI link. These methods can be useful when an app is dealing with different kinds of contexts and needs to display information about them without having to access individual fields within the context objects. For example, a calendar app can simply call *shortDescirbe* on context objects to get their descriptions to be displayed in an agenda view.

The abstract method *checkFields* is used to perform field-checking over the context object. The method checks if the fields are properly initialized and if the information they contain can be safely added to the Context Database. Apps can use this information to check information in a context object for correctness.

The *ContextData* class implements the *Parcelable* [83] interface which is the preferred method for marshaling data across processes in Android.[2] The Parcelable interface and its associated methods are inherited by all context classes by extending the *Context* class. This allows all context objects to be efficiently marshaled across processes through the Parcelable implementation provided by the Android system.

The abstract methods declared in the *Context* class need to be implemented in every concrete context class that extends it. Since the methods are context-specific, every context type will have a different implementation of the methods based on the functionality and fields contained within that context class.

7.3.3 Extending Contexts

Extending contexts is achieved through Java's object-oriented concepts [84] which allow one class to extend another class. Context classes extend existing classes and behavior by extending the respective Java class. For example, Listing 7.3 shows the *Movie* class. The *Movie* class extends the *Event* class, which corresponds to the *Movie* context extending the *Event* context. The fields title, date/time, location, contacts and URI are inherited by *Movie* from the *Event* class. The *Movie* class declares the fields for ticket and seat information which are not present in the *Event* class.

The hierarchical structure of contexts is similar to Java's object model, where each class has a common ancestor called *Object*. Every class can only extend one other class which keeps the class organization simple and prevents the problems associated with multiple inheritance. If a use case requires the use of multiple inheritance in relation to contexts, one possible solution is to extend one parent and to embed the other parents as sub-contexts.

[2]Parcelable and Serialization are implementations used for marshaling and unmarshaling Java objects. Android's Parcelable implementation allows objects to read and write from Parcels (data object) which can contain flattened data inside message containers, which offers better performance results on constrained platforms like Android.

```java
package msc.prototype.context;

import android.net.Uri;
import Java.util.Date;
import android.os.Parcelable;

class MovieData extends EventData {
  String ticket;
  Stirng seats;
  // other methods and parameters
}
```

Listing 7.3: Schema for *Movie* Context extended from *Event* Context

```java
void handleEventContext(EventData event) {
  // do something with event
}

void foo() {
  // movie context object
  MovieData movie = new MovieData();
  // pass handleEventContext a movie object
  handleEventContext(movie);
}
```

Listing 7.4: Schema for *Event* Context

7.3.4 Generalization of Contexts

Generalization of an object to its parent class means interpreting an object of the derived class as if it were an object of the parent class. This increases the re-usability of code as the same code can handle multiple types of objects based on how they are derived from a particular context. This allows creation of apps that target contexts situated higher in the hierarchy and work with all contexts directly below it. In terms of implementation using Java classes, this allows apps to include code written to work with one context, which will work with all contexts that have extended the specified context. Therefore, functions written for *Event* objects will also work with *Movie* objects as the system generalizes the *Movie* object into an instance of the *Event* object. In Java terminology, this is called *type casting.* Here the *Movie* object is cast as an *Event* object by the system.

Listing 7.4 shows a function *handleEventContext* that accepts *Event* objects. Another function `foo` calls `handleEventContext` and passes it a *Movie* object. The system automatically casts the *Movie* object to an *Event* object before passing it to the `handleEventContext` function.

Apps such as Calendar that primarily handle *Event* contexts can provide the same

```java
package msc.prototype.context;

class LocationData extends EventData {
  protected Stirng placename;
  protected double xpos;
  protected double ypos;
  // other methods and parameters
}
```

Listing 7.5: Schema for *Event* Context

```java
void useLocation{EventData event} {
  // gets location's / place name
  doSomething(event.getLocation().getPlaceName());
}

void updateLocation(EventData event) {
  // sets event's location to current location
  LocationData location = getCurrentLocation();
  event.setLocation(location);
}
```

Listing 7.6: Using *Location* context

services for all contexts that extend the *Event* class. This means contexts such as *Movie*, *Lunch*, *Meeting*, and *Concert* that extend the *Event* class are automatically converted to *Event* context objects before being used by the Calendar app. This allows the Calendar app to handle all types of events while writing minimal code and focusing on the features provided rather than putting in effort to support more context types.

7.3.5 Embedding Contexts

Embedding contexts is achieved by including the sub-context as an object reference inside the context class. This allows the embedded context to be referenced as a field. Listing 7.5 shows the *Location* class which is embedded in the *Event* class.

The *Location* class contains the fields and methods necessary to represent a *Location* context. By embedding the context in the *Event* context, every *Event* object will contain a reference to a *Location* object. This allows a context object to contain references to associated information even if the information itself may not be stored collectively with that context. The *Location* reference can be used to modify the location information associated with the *Event*, or to point it to another existing *Location* context. Listing 7.6 shows how the *Location* reference can be used to access the location information, and how different locations can be associated with an *Event*.

```
create table if not exists LOCATION(
 _id INTEGER PRIMARY KEY AUTOINCREMENT,
 place TEXT NOT NULL,
 xpos REAL NOT NULL,
 ypos REAL NOT NULL
);
```

Listing 7.7: Table schema for Location context

7.4 Context Database using SQLite

7.4.1 Initializing the database

The implementation for the Context Database uses SQLite version 3.7.11 as the backend for storing the contextual information. This version of SQLite is pre-installed on all Android v4.4.4 KitKat devices.[3]

The creation and initialization of the SQLite database was done through a separate app. The app creates a database called *Contexts.db* and initializes the necessary tables required for creating the Context Database. The Content Provider interface used to interact with the SQLite backend is also initialized through this app. The Content Provider is utilized by the Context Manager to interact with the Context Database through a set of standard API's.

7.4.2 Context entries and tables

Every context type has its own distinct table in the SQLite database. Each field in the Context Definition schema is stored as a separate column in the table. The *id* field defined in the abstract *Context* acts as the primary key in all context tables. This ensures that all context entries stored in a table are unique with respect to the contextual information they represent. The table schema for the *Location* context is shown in Listing 7.7.

Where a context extends another context, only the fields declared in the extended context are stored in extended context's table. The fields inherited from the parent context are stored in the parent context's table. For example, Listing 7.8 shows the table schema for the *Event* context and Listing 7.9 shows the table schema for the *Movie* context. The table for *Movie* contains a reference to an entry in the *Event* context where the inherited fields are stored. Both entries share the same id belonging to the *Movie* context added to the database. While inserting a *Movie* context, it is ensured that there is no clash or overlap with an existing entry in the *Event* table.

[3]Some devices are shipped with a different SQLite version as manufacturers install an updated version on the device. Version 3.7.11 is the version of SQLite binary shipped in the KitKat release dated 01-Nov-2013.

```sql
create table if not exists EVENT(
 _id INTEGER PRIMARY KEY AUTOINCREMENT,
 title TEXT NOT NULL,
 date INTEGER NOT NULL,
 uri TEXT NOT NULL);
);
```

Listing 7.8: Table schema for *Event* context

```sql
create table if not exists MOVIE(
 _id INTEGER PRIMARY KEY AUTOINCREMENT,
 ticketID TEXT NOT NULL,
 seats TEXT NOT NULL,
 FOREIGN KEY(_id) REFERENCES EVENT(_id)
);
```

Listing 7.9: Table schema for Movie context

The separation of fields into different tables allows separate access to contextual information. When an app requests contextual information related to *Events*, only the fields in the *Event* table are returned. Since the table for *Event* also contains entries from all contexts that have been extended from the *Event* context, all the contextual information related to events is accessed from a single table. Whenever an app requests information about *Movie* contexts, the entries from the *Movie* table are matched with their corresponding entries from the *Event* table before being returned. This allows efficient use of joins and multiple queries as a join is performed only when returning extended contexts, whose entries in the table are always less than those for the parent context.[4]

Whenever a context embeds another context, this relation is stored in a separate table unique to the relation between the two contexts. For example, the relation between an *Event* entry and a *Location* entry is stored in a separate table whose schema is shown in Listing 7.10. Each entry in this table depicts a relation between an *Event* entry and a *Location* entry.

7.4.3 Handling Duplicates in Database

Contextual information should not be duplicated in the Context Database. Since context entries represent contextual information, it is necessary to ensure that there is only one instance of a context in the Context Database. Two context entries are said

[4]Since every entry in the extended context's table has a corresponding entry in the parent context's table, the parent context's table will always contain at least the same number of entries as the extended context. Therefore, the number of entries in the parent context will always be greater than or equal to the number of entries in any extended context's table.

```sql
create table if not exists EventxLocation(
 e_id INTEGER, l_id INTEGER,
 FOREIGN KEY(e_id) REFERENCES EVENT(_id),
 FOREIGN KEY(l_id) REFERENCES LOCATION(_id)
);
```

Listing 7.10: Table schema for EventxLocation that stores the reference of Location objects within Events

to be duplicates if the contextual information they convey is the same.[5] The criteria for deciding whether two contexts are duplicates depends on the fields in the context, and is different for every context type. For example, when considering two *Event* contexts, if both entries have some fields that are the same, they may not necessarily be duplicates. But if a combination of multiple fields such as title, date/time and location are the same, the context can be considered to represent duplicate information.

Every time a context entry is to be inserted, it is first checked if it is a duplicate of an existing context in the Context Database. If no duplicates are found, the context entry is successfully inserted and given a unique *id*. If a duplicate is found, the Contextual Data Sharing Model can work with the following two options -

1. *Refuse the insert*: The Context Database refuses the insert operation and the context entry is not added to the database. A *duplicate context entry* error message is sent to the corresponding Context Manager along with the *id* of the original context entry in the Context Database. This option forces the Context Manager to explicitly query the Context Database for the original context and substitute the duplicate context with the original context's *id*. In effect, it discards the duplicate context, and returns the original context.

2. *Accept the insert and return the original id*: The Context Database accepts the insert operation but does not add the duplicate entry to the database. It returns the *id* field of the original context stored in the database as the *id* of the inserted entry. The Context Manager receives an *insert successful* message and sets the *id* of the inserted context object with the *id* of the original context object in database. In this option, the Context Manager is not aware that the insert option was related to a duplicate in the database, and considers the context entry to be successfully inserted in the database.

In both cases, there can be loss of contextual information in the fields not being used to check for duplicates between two context entries. If these fields are ignored or

[5]Similar contextual information does not necessarily correspond to actual matching of the information fields. For example, two events are taking place at two locations with the same co-ordinates, but different place-names. In this case, the two location contexts can be considered as overlapping each other even though the place-names are different.

overwritten, it results in an unwanted change in contextual information. For example, an *Event* entry being added to the database identified as being a duplicate of an existing entry. This means that the title, date/time and location field match an existing entry in the database. The other information in the context entry such as the URI field may not be checked at all. In this case, if the *id* of the context entry in the database is returned, it will create a discrepancy in the representation of contextual information in context objects with the same id. Some instances of this context will have the URI field from the Context Database, and others may have a different URI field. This is undesirable when dealing with contextual information. If this context is again saved to the Context Database, the URI field will be overwritten with the new information. This can lead to an accidental loss of contextual information.

To prevent such accidental loss of information and overwriting, duplicate context entries with other differing fields should be handled through an *update* operation. The Context Manager is made aware that a duplicate exists in the database with some differing fields and has the option of discarding the differences and updating the information in the database or accepting the differences and changing the information in the context being added to the database. This can still lead to loss of information and overwriting, but the Context Manager and the app will be aware of the exact contextual information being added to the database. This approach also prevents discrepancies in context objects with the same id since all information contained within such objects will match with the information stored in the Context Database.

7.4.4 Deletion Policy

The deletion policy dictates the deletion of contexts in order to keep the performance of queries within acceptable limits. The size of database t for which the deletion policy is triggered is highly dependent on the specific device configuration and usage of the Context Database. The impact of a table's size on the execution of queries is discussed in the Metrics and Performance chapter.

For the deletion policy we considered using paging algorithms [85, p. 288] such as First-in First-out (FIFO), Least frequently used (LFU) and Least recently used (LRU) to decide which contexts are to be deleted. Using LFU and LRU requires storing a timestamp every time a context entry is accessed in order to determine its frequency of use. This requires a write to the database every time a context entry is read, which is unfeasible when using SQLite as it severely affects the speed of read operations.[6] Therefore, the implementation uses FIFO to select contexts for deletion since using it

[6]SQLite does not support storing read timestamps like other RDBMS such as MySQL. In order to implement this functionality in SQLite, an explicit write operation to the table is necessary with every read operation. This locks the table during the write operation, and prevents concurrent read access, thereby affecting the performance of the query.

does not require storing timestamps on read operations.

FIFO orders entries based on the time when they were added to the database. This can be implemented by storing a timestamp with every entry inserted to the database. The timestamp is updated whenever the entry is updated to denote the change in information. In FIFO, the first n entries are selected for deletion based on their timestamps. The value of n is selected so as to satisfy the condition $(0 < n < t)$.

When the value of n is small, the value of $(t-n)$ is comparable to the value of t. This causes the database size to grow to the threshold size sooner and causes degradation of performance as the deletion policy is triggered more often. If the value of n is large, the value of $(t - n)$ is comparably smaller than t, and the database size reaches threshold size more slowly. The deletion policy is triggered after significant time intervals, and prevents the degradation of performance over time. However, this also deletes more contexts in a single operation and may delete contextual information that could have been useful.

The value of n is calculated based on a combination of conditions such as device configuration, frequency of queries and operational costs involved.[7] In this implementation, the value of n is within the range $(1000 < n < 5000)$ depending on the context table under consideration.[8]

Along with FIFO, context entries for some context types can also be selected based on the relevance of their contextual information. For example, some context types such as *Event* are based on a specific date/time, which is more relevant than the insertion timestamp used in FIFO. Such context types are deleted based on relevance using the ordering of information present in the context table.

If a context entry being deleted is an embedded entry, then it is only deleted when all of its parent context entries are also marked for deletion. If a context entry being deleted is an extended entry, or contains a relation to an extended entry, then it is ensured that all related fields in different tables are also deleted. This ensures that useful data is not deleted in the case of embedded entries, and that all related data is removed together in the case of extended entries.

7.5 Context Manager as a Static Java Class

The Context Manager is instantiated as a static Java class in the user app's process, and is responsible for all interactions between the app and the Context Database. It interacts with the Content Provider interface to request queries from the Context

[7] Creating such policies requires further extensive testing and is not within the scope of this research.

[8] Different context tables can have different values of n, as each context table stores different contextual information based on relevance and frequency of use. For example, location entries will be used differently than contacts, and therefore can have a different value of n.

Database and interprets the response in a format understood by apps. It also performs operations such as error-checks and marshaling on context objects. All operations performed by the Context Manager are executed in the user app's process.

Listing 7.11 shows the implementation of the Context Manager using a static Java class called *ContextManager* that refers to the Content Provider of the Context Database using the URI `content://msc.prototype.contextprovider.cp/` that refers to the Content Provider used to access the Context Database. The application's Content Resolver is used to resolve the URI to the Content Provider and to receive responses within the the app. When inserting or updating a context object, the Context Manager checks if the object has its *id* field set, which indicates that the object exists in the Context Database. It also performs checks for errors and field-completeness and proceeds only if no errors are found. When an app requests context objects from the Context Database, the Context Manager forms the appropriate query and sends the request to the Content Provider. Upon receiving results, it creates the corresponding context objects before returning them to the app.

The objects created by the Context Manager are tied to the app that holds the instance of the Context Manager. The objects are instantiated in the apps' data space and their lifetime is tied to the apps' lifetime.[9] Whenever the system performs garbage collection, the context objects are automatically cleaned without requiring any form of memory management. Due to the restrictions placed by sandboxing, each app can access only its own data, and therefore only those context objects that are created in its data space.

7.6 Usage by Apps

7.6.1 Providing Context Classes in an Android Library

Apps use the Contextual Data Sharing Model by including a library situated in a file called `ContextManager.aar`, an Android Archive module[10] which should be copied to an app's library (*lib*) directory along with adding a reference to the app's Gradle build.[11] Listing 7.12 shows the gradle build for an app using the Contextual Data Sharing Model by including the library file as an external library. The library contains the context

[9]In Android, objects instantiated by an app are created in the *heap*, a dynamic storage memory provided to the app.

[10]An Android Archive (extension .AAR) module is compiled from a Library project and is meant to be shared as a module between projects. It is a Zip archive containing a combination of compile code such as JAR and/or native .so files along with resources such as manifest, assets and other ancillary files.

[11]Gradle is a project automation tool that provides dependency management features similar to Apache Ant and Apache Maven. Gradle is the default build system in Android studio, the official IDE for Android application development. More info at https://developer.android.com/tools/studio/index.html

```java
package msc.prototype.context;

public class ContextManager {
  static final String uri_Event = "Event";
    static final String uri_authority =
      "content://msc.prototype.contextprovider.cp/";

    public static int insert(ContentResolver resolver, EventData event) {
        Uri uri = Uri.parse(uri_authority + uri_Event);
        ContentValues values = new ContentValues();
        // insert all event fields into ContentValues
        values.put("eventtitle", event.getEventname());
        . . .
        // check if event already has an id
        // check if event's subcontexts have an id
        . . .
        // perform error-checks on event
        . . .
        // insert event to Context Database
        Uri response = resolver.insert(uri, values);
        // parse the response and set event's id
        . . .
        return 0;
    }

    public static ArrayList<EventData> getEvents(ContentResolver resolver) ↩
        {
        Uri uri = Uri.parse(uri_authority + uri_Event);
        Cursor result = resolver.query(uri, null, null, null, null);
        ArrayList<EventData> events = new ArrayList<>(result.getCount());
        result.moveToFirst();
        // initialize event objects from result and add to events
        . . .
        events.add(event);
        result.close();
        return events;
    }
}
```

Listing 7.11: Implementation of Context Manager as a Static Java Class

classes, the Context Manager class and implementation of other APIs required for using the model. The context classes and the Context Manager class are packaged and used as a external library since these classes cannot be installed at a system level on the Android device.[12] The context classes cannot be installed on an unmodified Android distribution due to the security model of the platform. In order to demonstrate the Contextual Data Sharing Model, the apps gain access to the context classes through the integrated library which contains a copy of these classes.

```
dependencies {
    compile(name:'ContextProvider', ext:'aar')
}
```

Listing 7.12: Adding the library as a module in Gradle

7.6.2 The *Change vs. New* Policy

Apps that insert context objects in the Context Database must be aware of instances where an existing context is accidentally updated instead of adding a new context to the Context Database. The *Change vs. New* policy informs developers of the correct use cases for insert and update operations in order to prevent unwanted changes to the contextual information present in the Context Database.

The following example demonstrates the basis and need for this policy: an app updates the co-ordinates within a *Location* context associated with an existing *Event* to reflect a change in the *Event's* location. Other contexts that are associated with or use this particular *Location* as a sub-context or through a reference will now point to the updated co-ordinates. As a consequqnce, any other *Event* contexts that share this textitLocation context will have their venue (location) changed. This results in a change in the contextual information and introduces incorrect information and affects the integrity of information utilized.

To prevent this, it is necessary for developers to be aware of the *Change vs. New* policy which dictates that *if a context is being updated, refined or corrected in some form, only then may the information within the context be changed or updated. In all other cases where information is being modified to change the intended meaning of the context, a new context object should be created and added to the database.* Applying this policy to the example described above tells the developer to create a new *Location* object since the intended meaning of the context is being changed.[13] Following this policy is

[12]System classes are those classes that are available to all apps and processes and do not need to be included with the apps' code. The app only needs to use the provided APIs to use the classes and their associated methods in its code.

[13]The meaning of a context is its intended contextual information. A *Location* context is used to store location information about a place. By changing its co-ordinates, the location and therefore the meaning of the context is being changed.

left to the discretion of the developers since it is impossible to check programmatically whether a change made to any context is incorrect or violates the principle behind the *Change vs. New* policy.

7.7 Demonstration of Apps using the Contextual Data Sharing Model

Here we demonstrate the use of the Contextual Data Sharing Model in the movie ticket booking use case. Fig. 7.2 shows a screenshot of the app that allows users to book movie tickets. Users select the movie they wish to watch along with the preferred date/time, theater and number of seats. The app then generates a ticket with the selected details and a ticket ID and seat information. It stores this information in a *Movie* context object along with a link to the movie's IMDb page. The app then inserts the information in the Context Database by calling the *insert* method of its Context Manager. A sample of the code used for performing the above steps in a movie ticket booking app is provided in Listing 7.13.

```java
// create context object with movie information
MovieData movie = new MovieData(
  moviename,date,loc,contactData,uri,ticket,seats);
// insert context object in Context Database through Context Manager
if(ContextManager.insert(getContentResolver(), movie) != 0) {
  Toast.makeText(getApplicationContext(),
    "ERROR INSERTING MOVIE CONTEXT",Toast.LENGTH_LONG).show();
}
```

Listing 7.13: Movie ticket booking app

The apps used in the demonstration were created with distinct packages and developer signatures so as to prevent any implicit data sharing between them. This ensured that each app was unaware of the identities and existence of the other apps and acted in isolation. All apps interacted with the Context Database through the Context Manager bundled within the app.

A calendar app that retrieves *Event* contexts from the Context Database will also receive the saved movie information along with the other events. The app will interpret the movie information as an *Event* context and would have access to all of the fields within the *Movie* context which are inherited from *Event*. This allows the app to change contextual information such as date/time and contacts associated with the movie context through user input. The calendar will provide features such as notifications and reminders for the movie similar to other event contexts. This will save the user the effort of entering the information and setting up reminders, as the app retrieves the needed information from the Context Database.

The messaging app shown in Fig. 7.3 allows users to insert contextual information in the message body. It provides users the option to select a context entry, which is retrieved from the Context Database. The app then allows users to select fields from the context which are then inserted in the message body. The app also provides the options of using the *Contacts* from a context object to populate the recipients field of the message. Using a messaging app that allows users to select contextual information saves users the trouble of finding and entering the information. It also helps them complete common tasks such as sending movie details to all attending contacts with a few clicks within the app.

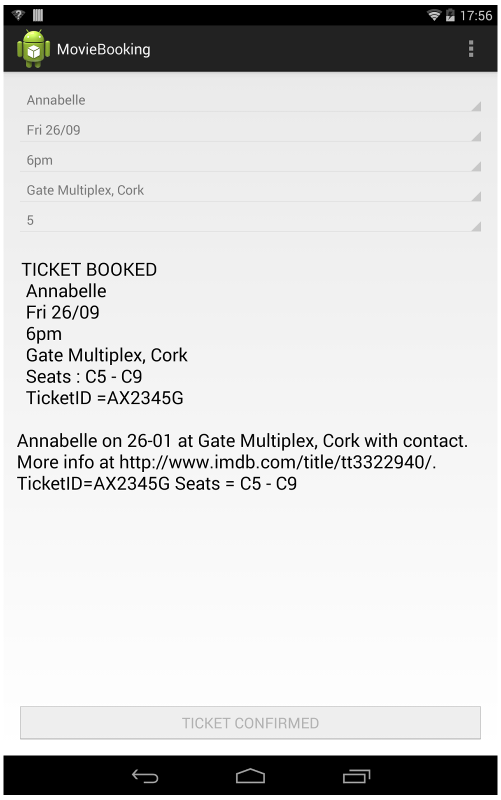
The maps app shown in Fig. 7.4 displays a list of upcoming events retrieved from the Context Database upon opening the app. Users select an entry by clicking on it, which tells the app to use the location field of the selected *Event* context as the destination for providing navigational features. Alternatively, users can enter the address in the input bar provided at the top to set the destination themselves. The maps app provides smarter navigational features by offering the user a list of destinations they are most likely to use. In the movie ticket booking use case, the app shows an entry for the movie in the list of events. This list is shown right after the user opens the app, which saves the effort of entering the address or selecting a location in the app in order to navigate to the theater.

The reminder app provides notifications based on the context entries stored in the Context Database. It retrieves the *Movie* context and sets a reminder triggered by the theater's location. Once at the theater, the app shows this notification as seen in Fig. 7.5 that displays the ticket and seat information to the user. The user does not have to open any app or search through notes and messages to access the ticket and seat information since it is easily accessible as a notification. To create such contextual reminders, the user does not have to explicitly enter the information required to create reminders since the app retrieves it from the Context Database.

The flow of information and the user interactions can be seen in Fig. 7.6, which depicts how the apps use the Contextual Data Sharing Model to create contextually aware services for the user.

7.8 Permissions and Security

Android is a privilege-separated operating system [86], in which each application runs with a distinct system identity based on the concepts of Linux user ID and group ID. Parts of the system are separated into distinct identities, which isolates applications from each other and from the system. This forms the basis of the sandboxing model.

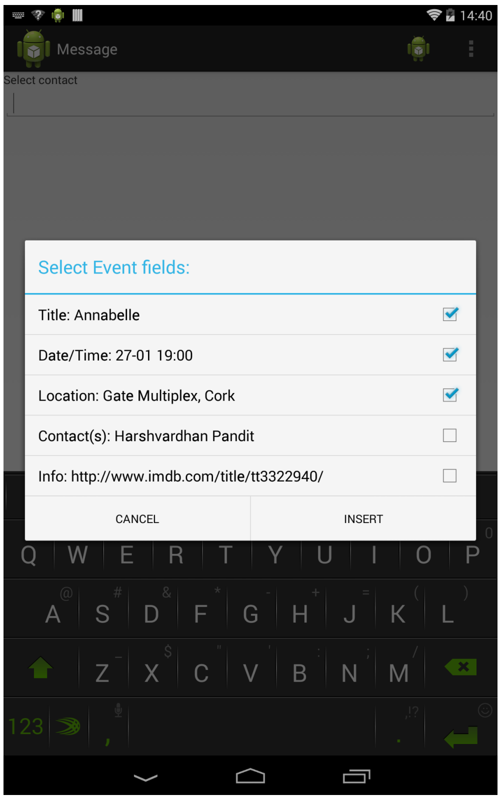


Annabelle
Fri 26/09
6pm
Gate Multiplex, Cork
5

TICKET BOOKED
Annabelle
Fri 26/09
6pm
Gate Multiplex, Cork
Seats : C5 - C9
TicketID =AX2345G

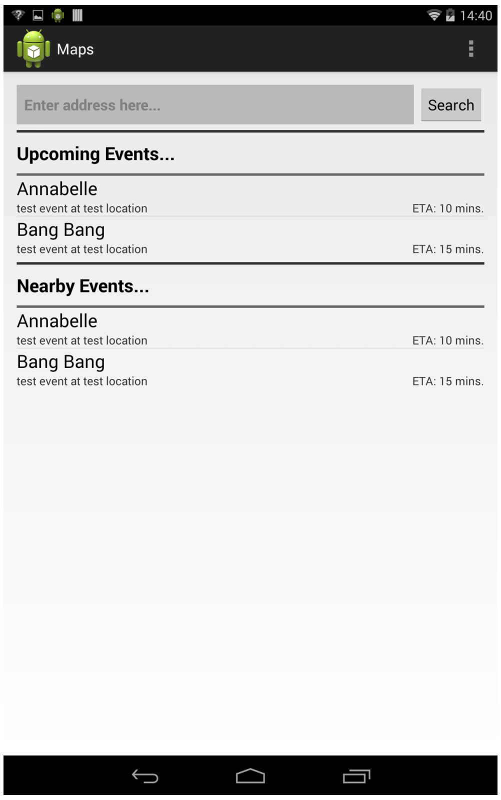
Annabelle on 26-01 at Gate Multiplex, Cork with contact. More info at http://www.imdb.com/title/tt3322940/. TicketID=AX2345G Seats = C5 - C9

Figure 7.2: Movie Booking app



Title: Annabelle	✓
Date/Time: 27-01 19:00	✓
Location: Gate Multiplex, Cork	✓
Contact(s): Harshvardhan Pandit	☐
Info: http://www.imdb.com/title/tt3322940/	☐

Figure 7.3: Messaging app



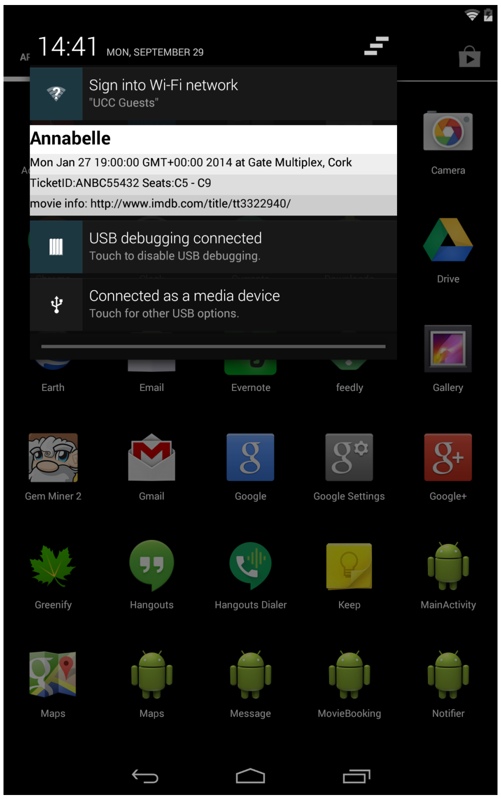
Upcoming Events...

Annabelle test event at test location	ETA: 10 mins.
Bang Bang test event at test location	ETA: 15 mins.

Nearby Events...

Annabelle test event at test location	ETA: 10 mins.
Bang Bang test event at test location	ETA: 15 mins.

Figure 7.4: Maps app



Sign into Wi-Fi network
"UCC Guests"

Annabelle
Mon Jan 27 19:00:00 GMT+00:00 2014 at Gate Multiplex, Cork
TicketID:ANBC55432 Seats:C5 - C9
movie info: http://www.imdb.com/title/tt3322940/

USB debugging connected
Touch to disable USB debugging.

Connected as a media device
Touch for other USB options.

Figure 7.5: Reminders app

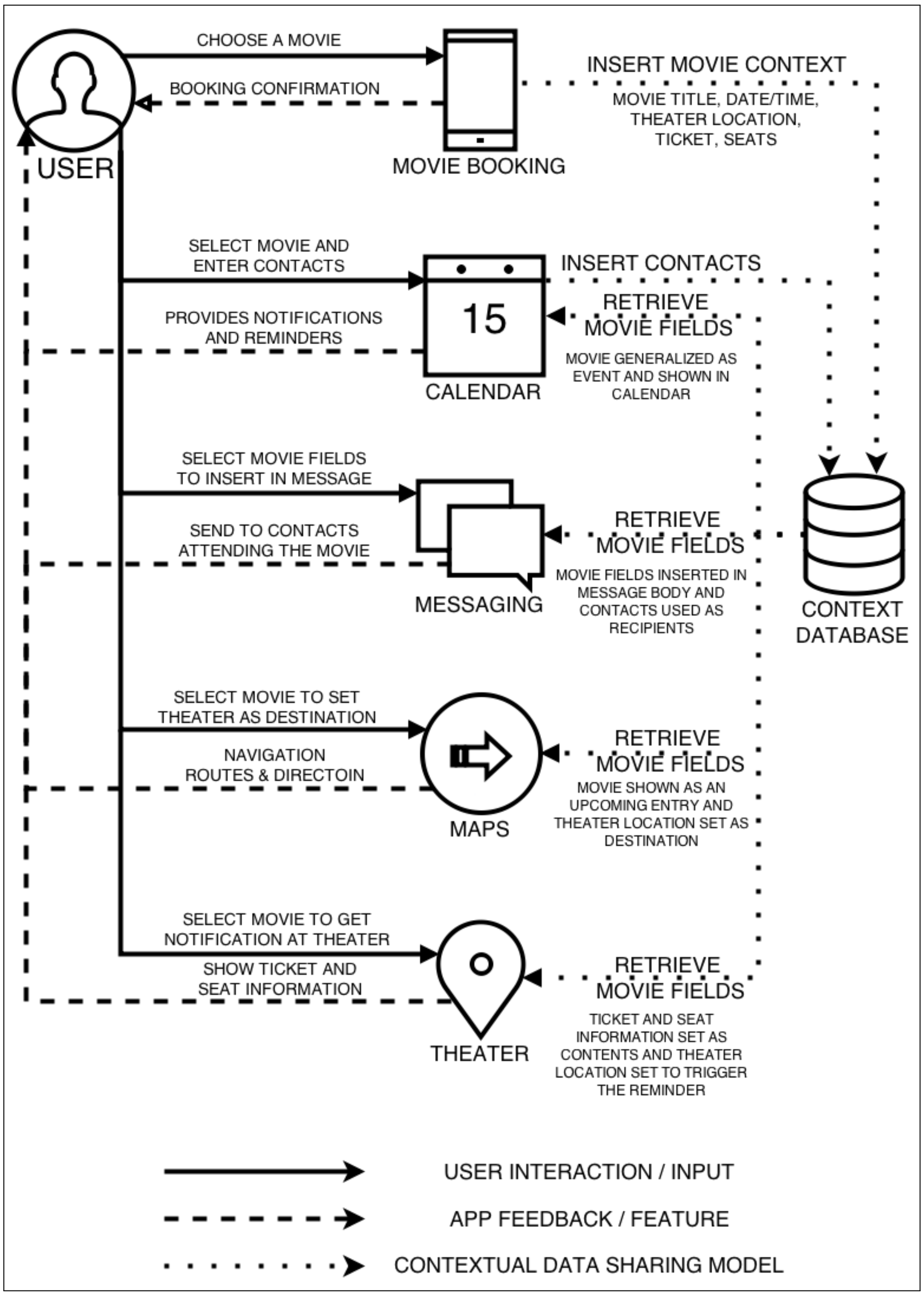

CHOOSE A MOVIE
BOOKING CONFIRMATION
USER
MOVIE BOOKING
INSERT MOVIE CONTEXT
MOVIE TITLE, DATE/TIME, THEATER LOCATION, TICKET, SEATS
SELECT MOVIE AND ENTER CONTACTS
PROVIDES NOTIFICATIONS AND REMINDERS
15
CALENDAR
INSERT CONTACTS
RETRIEVE MOVIE FIELDS
MOVIE GENERALIZED AS EVENT AND SHOWN IN CALENDAR
SELECT MOVIE FIELDS TO INSERT IN MESSAGE
SEND TO CONTACTS ATTENDING THE MOVIE
MESSAGING
RETRIEVE MOVIE FIELDS
MOVIE FIELDS INSERTED IN MESSAGE BODY AND CONTACTS USED AS RECIPIENTS
CONTEXT DATABASE
SELECT MOVIE TO SET THEATER AS DESTINATION
NAVIGATION ROUTES & DIRECTOIN
MAPS
RETRIEVE MOVIE FIELDS
MOVIE SHOWN AS AN UPCOMING ENTRY AND THEATER LOCATION SET AS DESTINATION
SELECT MOVIE TO GET NOTIFICATION AT THEATER
SHOW TICKET AND SEAT INFORMATION
THEATER
RETRIEVE MOVIE FIELDS
TICKET AND SEAT INFORMATION SET AS CONTENTS AND THEATER LOCATION SET TO TRIGGER THE REMINDER
USER INTERACTION / INPUT
APP FEEDBACK / FEATURE
CONTEXTUAL DATA SHARING MODEL

Figure 7.6: Information flow in apps using the Contextual Data Sharing Model

Additional finer-grained security features are provided through a ‘permission’ mechanism that enforces restrictions on the specific operations that a particular process (app) can perform, and per-URI permissions for granting ad-hoc access to specific pieces of data.

Android’s native implementation of permissions can be utilized to control access to contextual information in the Context Database. This requires creating new permissions that specify the context types the app wishes to use and the operations to be performed on it. The permissions required by the app to use contexts can be declared in the apps’s manifest in a manner similar to existing permissions in Android.

Listing 7.14 shows an example of how permissions can be used to declare the use of contexts by an app through its manifest. Apps must declare Read/Write permissions for the context types they wish to use. If a context contains sub-contexts in its definition, then the system can automatically grant the related permissions for using the sub-contexts, or force the developer to explicitly specify those permissions in the manifest. The choice is based on the implementation of permissions enabling the developer to select either of the two courses. It is recommended to specify all the required permissions explicitly in the manifest since this is the approach followed by the existing implementation of permissions on Android.

```xml
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="http://schemas.android.com/apk/res/android"
    package="com.example.android.demo"
    android:versionCode="1"
    android:versionName="1.0" >

  <uses-permission android:name="msc.prototype.context.EVENT.READ"/>
  <uses-permission android:name="msc.prototype.context.EVENT.WRITE"/>

  <!-- related permissions
    msc.prototype.context.LOCATION.READ
    msc.prototype.context.LOCATION.WRITE
    msc.prototype.context.CONTACT.READ
    msc.prototype.context.CONTACT.WRITE -->
</manifest>
```

Listing 7.14: Context permissions in the app manifest

The use of permissions allows identification of what contexts an app is using. This can be used to analyze the amount of contextual information being used by the apps, which is helpful to recognize apps that steal information. The permissions are also necessary to prevent malicious access by apps, which can corrupt information in the Context Database. The permissions denote the use of information requested by the app.

This information can be shown while installing the app to get an informed consent from the users.

7.9 Summary

The examples and the demonstrations of the use cases show how contextual sharing of information is achieved by sharing information through the Context Database. The apps used the contextual information stored in the Context Database to provide contextual services, which allow the user to enter significantly less information in various apps as the necessary information was retrieved from the Context Database. This resulted in the users getting easier access to required features and allowed them to complete their tasks faster. The use of Context Database prevents the duplication of effort and information by storing data in a common and accessible way. The demonstration validates the usefulness of the Contextual Data Sharing Model.

8

Performance Evaluation

"Beware of bugs in the code; I have only proved it correct, not tried it."
– Donald Knuth

8.1 Motivation

The Contextual Data Sharing Model allows apps to retrieve, store and share contextual information by using the Context Manager to interact with the Context Database. Apps can use the contextual information to fill the UI elements shown to the user for interacting with the app, which creates a relation between the Contextual Data Sharing Model and the app's user experience.[1] Therefore, the performance of the Contextual Data Sharing Model and its individual components have an impact on the performance of apps using the model, and by extension on the user experience when using such apps. This chapter provides comparisons, performance tests and experiments that are used as a proof-of-concept demonstration to gauge the feasibility of the Contextual Data Sharing Model and decide operating parameters that affect the performance and user experience of apps.

8.2 Testing Parameters

To test the efficiency of the model, the prior movie ticket booking use case was compared in terms of duplication of information and user effort. The performance of the Contextual Data Sharing Model consists of the performance of individual components

[1]Compared to tasks carried out in the background such as downloading a file, UI elements and the information they contain affect the user experience in a larger capacity since the user directly sees and interacts with them.

such as the Context Database and Context Manager. Some operations in the performance tests such as IPC and marshaling that lie outside the Contextual Data Sharing Model are handled by the Android system.

Performance is primarily based on the time taken to execute the various operations, where the time for each operation is broken down to reflect individual times for the different components. The time taken by the Context Database consists of the time required to receive and parse a query, the time required to execute the query and the time required to create appropriate responses. The time taken by the Context Manager consists of the time required to create the appropriate query, the time required to send it to Context Database for resolution and the time required to interpret the received response and convert it into context objects. Supplementing these operations is the IPC mechanism handled by the Android system which manages the sharing of data between the Context Database and the Context Manager.

All tests were performed on the *Event* table that was populated with random entries at the time of database initialization.[2] The corresponding sub-contexts *Location* and *Contacts* were also randomly populated.

Since the tests are based on the execution of queries and the interpretation of results, the number of entries (records) in the Context Database have a significant impact on the time required to complete each operation. Each tests was run multiple times (*n=100*) and for different number of records *(100, 500, 1000, 5000, 10000, 50000, 100000)* in the *Event* table, with only those queries that successfully completed execution being considered for evaluation.

The time taken to execute a query is depicted by t, with t_{min} depicting the minimum value, t_{max} depicting the maximum value, t_{avg} depicting the average, and t_{stdev} depicting the standard deviation.

8.3 Operating Environment

The experiments were carried out on a *Nexus 7* running Android v4.4.4 Kit-Kat.[3] No other software or apps were installed and the stock ROM was not modified, rooted or supplemented in any way. The device battery was kept at near full charge, and the connectivity options (Wi-Fi, Bluetooth, NFC) were disconnected to prevent unwanted interferences in the tests. The performance metric values were obtained through timestamps from logging at strategic points in the code. The logs and process stats were

[2]Using Java's Random class (http://docs.oracle.com/javase/7/docs/api/java/util/Random.html) that generates a stream of pseudo-random numbers using a 48-bit seed modified using a linear congruential formula.

[3]Nexus 7 Wi-Fi only, model year 2013, Qualcomm Snapdragon S4Pro, CPU Quad-core 1.5 GHz Krait, GPU Adreno 320, 2 GB RAM.

gathered using *adb* and logcat[4] interfaces.

8.4 User Experience Comparison

Table 8.1 shows the information sources utilized by the apps as they carry out the various tasks related to the movie ticket booking context. The label *APP* denotes information generated by the app, *USER* denotes information entered by the user for the first time, and *GET* denotes information retrieved from the Context Database. Columns contains *Movie* fields and rows contain apps. The apps retrieve the movie fields from the Context Database which saves the user the effort of entering the information into the app. Compared to the number of times the user had to enter information in apps in Table 1.1, the user has to enter the information only the first time as the apps retrieve the stored information from the Context Database.

Table 8.1: Information sources used by apps utilizing the Contextual Data Sharing Model in the movie ticket booking use case

App used	**Movie Title**	**Show Date/Time**	**Attending Contacts**	**Theater Location**	**Ticket Info**
Booking	*USER*	*USER*	*NA*	*USER*	*APP*
Calendar	*GET*	*GET*	*USER*	*GET*	*GET*
Messages	*GET*	*GET*	*GET*	*GET*	*GET*
Maps	*GET*	*GET*	*GET*	*GET*	*GET*
Reminder	*GET*	*GET*	*GET*	*GET*	*GET*

A comparison of the effort made by the user can be made using Table 1.1 and Table 8.1 to count the number of steps taken by the user in each app with and without the use of the Contextual Data Sharing Model. It can be clearly seen that when apps use the Contextual Data Sharing Model, the user only needs to select the available information instead of entering it in the app. The effort required to select information already available can be considered to be less than the effort required to enter information by typing it or copying it. The demonstration of the model also shows features such as showing upcoming events to select the destination of routes in maps that were not possible earlier.

[4] Android Debug Bridge (adb) is a versatile command line tool that allows debugging a connected Android-powered device. Logcat provides a mechanism for collecting and viewing filtered logs from various applications and portions of the system through adb. More info about adb and logcat can be found at https://developer.android.com

8.5 Performance of Context Database

8.5.1 Inserting an *Event* entry in Context Database

Table 8.2 shows the time required to insert one *Event* entry in the Context Database where the total time includes the time required for unmarshaling values, inserting *Contact, Location,* and *Event* entries in their respective tables, creating relational entries in various tables, and checking for duplicates.

Table 8.2: Time required to insert one *Event* entry into Context Database

Entries	t_{min}(ms)	t_{max}(ms)	t_{avg}(ms)	t_{stdev}(ms)
100	1	5	1.65	0.81
500	1	5	1.72	1.02
1000	1	5	2.03	1.14
5000	2	9	4.21	1.60
10000	4	19	6.00	2.28
50000	9	49	12.19	6.51
100000	20	119	42.55	12.08

8.5.2 Retrieving *Events* from Context Database

Table 8.3 shows the time required to retrieve all *Event* entries from the Context Database where the query executed the required operations to performs joins over *Event, Contact, Location* and their relational tables to retrieve all associated information in a single row of the result.

Table 8.3: Time required to retrieve *Event* entries from Context Database

Entries	t_{min}(ms)	t_{max}(ms)	t_{avg}(ms)	t_{stdev}(ms)
100	1	5	1.87	0.87
500	1	5	2.11	0.94
1000	1	5	2.34	1.13
5000	3	10	5.42	1.95
10000	9	39	12.05	4.29
50000	10	49	15.83	6.98
100000	20	137	31.51	15.73

8.6 Performance of Context Manager

8.6.1 Inserting an *Event* entry through Context Manager

Table 8.4 shows the time required to insert one *Event* context through the Context Manager where the total time includes the time required to perform error and validation checks, the IPC between user app and the Content Provider,[5] checking for duplicates, and inserting the entry in the Context Database.

Table 8.4: Time required to insert one *Event* context through Context Manager

Entries	t_{min}(ms)	t_{max}(ms)	t_{avg}(ms)	t_{stdev}(ms)
100	1	25	1.70	3.10
500	2	47	2.90	5.01
1000	5	58	6.77	7.41
5000	25	243	35.80	34.02
10000	51	491	62.34	48.86
50000	74	783	89.02	89.18
100000	100	1176	130.20	137.60

8.6.2 Retrieving *Events* through Context Manager

Table 8.5 shows the time required by the Context Manager to retrieve all *Event* entries from the Context Database where the total time includes the time required to execute the database query, perform the IPC between Content Provider and user app, and instantiate *Event* objects in the user app's data space.[6]

Table 8.5: Time required to retrieve *Event* contexts through Context Manager

Entries	t_{min}(ms)	t_{max}(ms)	t_{avg}(ms)	t_{stdev}(ms)
100	10	39	16.72	4.60
500	100	198	120.61	10.05
1000	200	388	240.95	16.28
5000	500	1781	663.56	115.86
10000	2000	4896	2265.74	364.46
50000	5000	10192	6681.94	712.68
100000	10024	18094	11459.12	1429.04

[5]The IPC and marshaling is performed by the Android system. Its time in included as part of the Context Manager's operations since it is part of the operation. The total time is useful to identify the performance of the Contextual Data Sharing Model from the user's point of view.

[6]The time taken by the Context Manager is inclusive of the time required to create the *Event* objects since the user app will receive the database results as *Event* objects from the Context Manager.

8.7 Comparison of Insert and Retrieval times

A comparison of the time taken by the insert and retrieve operations of the Context Manager is given in Fig. 8.1. The graph is plotted by comparing the range of values containing the minimum (t_{min}), maximum (t_{max}) and average time (t_{avg}) required for an operation with the number of records in Context Database.

Observations

Observing the graph and the associated tables leads to the following points:

1. The time required for inserting contexts is comparably less than the time required for retrieving contexts;
2. There is substantial deviation between the minimum and maximum values;
3. The standard deviation is closer to the minimum values, which signifies better average performance;
4. The increase in time is almost linear compared to the increase in number of records;
5. There is a sudden increase in the values at around $n=500$;[7]
6. For most of the time, the average is closer to the minimum values;
7. The average retrieval time reaches $t=120msataboutn=100$.

Conclusions

The observations lead to the following conclusions:

1. The minimum and average insertion values at large database sizes are within an acceptable range for responsiveness in UI (0-100ms [87]);
2. The maximum values show the extent of fluctuations possible during operation;
3. The number of entries returned in results should be restricted to around $n=100$ to keep the time within an acceptable range.

[7]The sudden increase in the time values can be attributed to the allotment of heap space by the Android system. If the results of a query do not fit in the app's available heap space, the system attempts to allot more heap space. This stalls the app process until the heap space is made available. Creating a large number of *Event* objects also fills up the heap space, and has a similar effect.

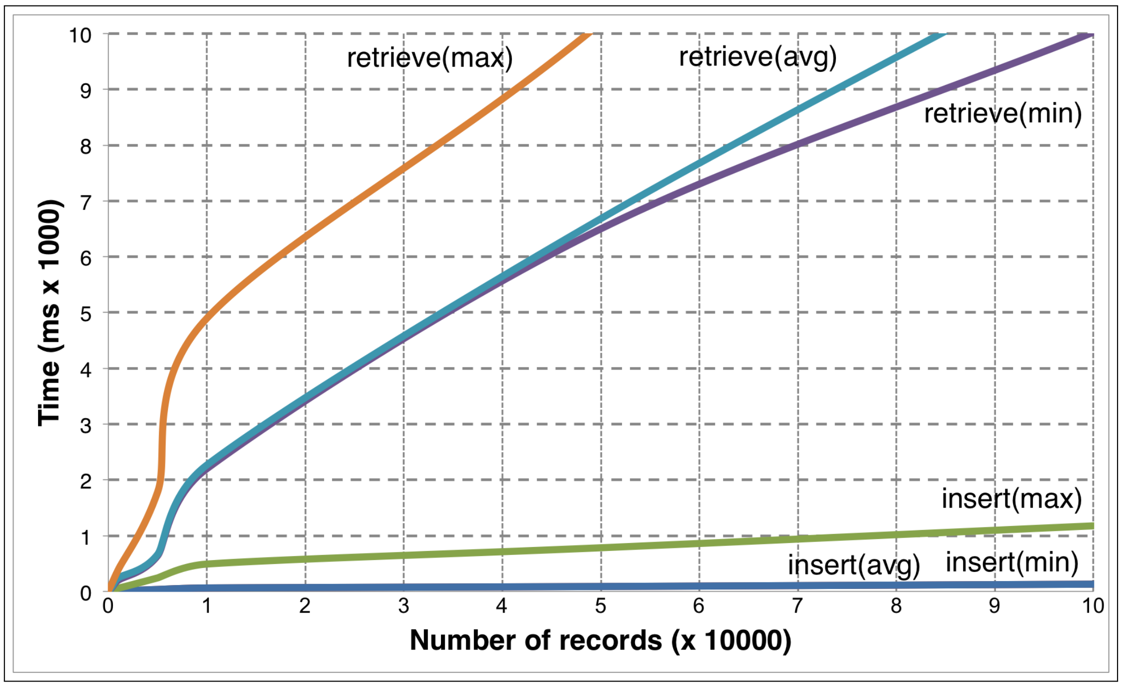


Figure 8.1: Comparison of insert/retrieval operation times

8.8 CPU load during retrieval operations

The impact of the Contextual Data Sharing Model and its operations on CPU performance can be seen in Fig. 8.2. It shows the CPU load for retrieving *Event* contexts through the Context Manager. The total number of *Event* entries retrieved from the Context Database was *n=10000*. The total time required for the operation is *12.5s*, and includes the time for UI interactions within the app used to run the test. In the figure, the horizontal axis depicts the run-time given in seconds, and the vertical axis depicts the CPU usage as a percentage. The retrieved *Event* contexts are displayed on screen using a List View.[8]

The various operations that take place at different time instances are:

- *t= 0s to 3.2s*: No activity. Some spikes in the CPU load can be attributed to the management of UI elements and memory by the system;
- *t= 3.2s*: UI interaction which sends a request to the Context Manager for all *Event* contexts. The Context Manager constructs the appropriate query and forwards it to the Context Database;

[8]A ListView is a view group that displays a list of scrollable items that are automatically inserted using an Adapter object that pulls content from a source such as an array or database query. More info about ListViews can be found at https://developer.android.com

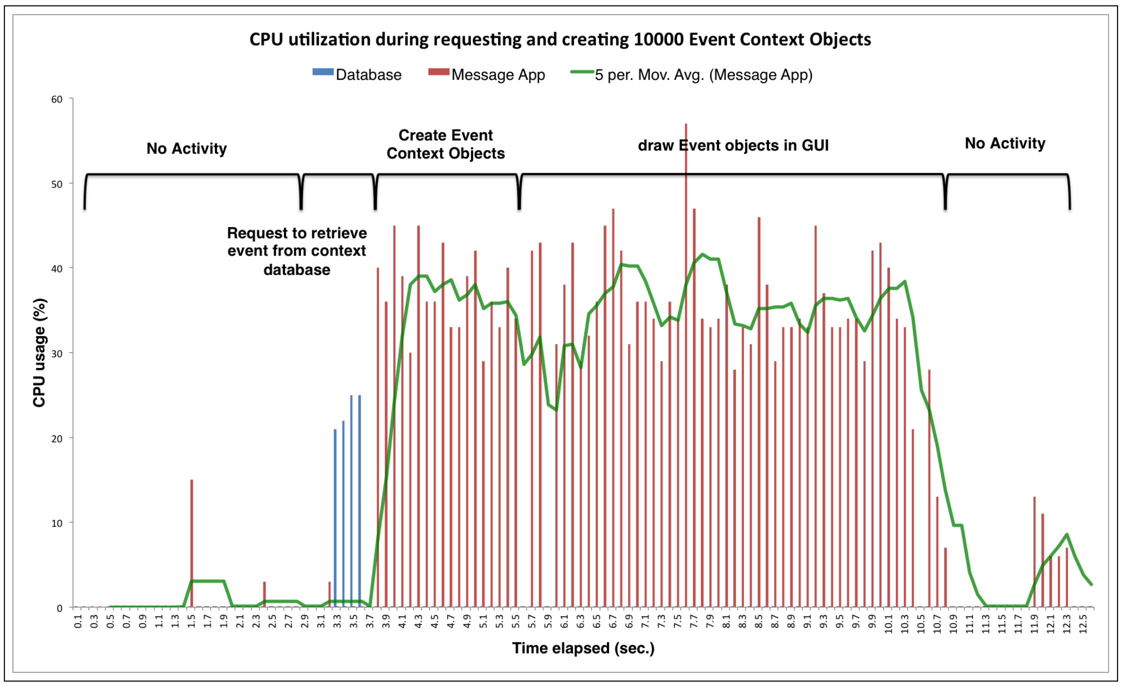


Figure 8.2: CPU load during retrieval operation

- *t= 3.3s to 3.6s*: The Context Database receives the request, parses it, and executes the query on the SQLite database to retrieve all *Event* entries. It then sends these back to the Context Manager;
- *t= 3.6s to 3.9s*: The time taken by the IPC mechanism to send the data from Context Database to Context Manager;[9]
- *t= 3.9s to 5.5s*: The Context Manager receives the entries and instantiates *Event* objects;
- *t= 5.5s to 10.7s*: The app receives the references to the *Event* objects and adds them to a ListView which is drawn on the screen;
- *t= 10.7s onwards*: UI interactions within the app such as closing the ListView and exiting the app.

The total duration of the operations from sending the request until displaying the results on screen is about *t=7.5s*, which is well outside the acceptable range for UI interactions. Most of this time is spent instantiating the objects and drawing them on screen. If the number of entries retrieved from the database is restricted to $n = 100$

[9]In Android, the IPC mechanism only copies the references to the data in memory between processes rather than the entire data. This saves the costs involved in copying all of the data and works within the sandboxing model.

using the analysis of Fig. 8.1, it would reduce the CPU time and memory required for the operation. This will allow the app to display the results in less time which allows for fluid user interactions. The average CPU load at all times is well below 50%, which can be considered as not being under stress. This allows the CPU to run other apps and operations in the background. If an app wishes to perform operations that may take significant time to execute and stall UI operations, Android provides various mechanism to execute operations on another thread to prevent freezing the UI. In this case, the retrieval operation takes a significant amount of time to complete, and therefore must be executed on a non-UI thread.

8.9 Summary

The performance of the Contextual Data Sharing Model depends on a number of factors, amongst which the number of records in the database is the major factor that affects the performance of the operations. The time required for executing insert operations is within an acceptable range for higher database sizes, while the time required for retrieving entries from the database crosses the range for responsiveness in UI at about n*=100.* Limiting the number of entries returned from a query to 100 will allow operations to complete in an acceptable time and provide better responsiveness in UI.

Further testing and experimentation of the various operations need to be performed over different use cases as apps use operations in different ways. The performance of multiple queries executed concurrently also needs to be tested. The number of apps using the Context Database and the Contextual Data Sharing Model can also affect the performance of operations. Devices differing in specifications and environments will also have an impact on the performance of the model. All these factors indicate the need for extensive future testing of the model in a range of use cases and devices.

Part V

This Research and its Future Potential

9 Conclusion

"The most important property of a program is whether it accomplishes the intention of its user."
– C.A.R. Hoare

The Contextual Data Sharing Model allows smartphone applications to utilize and share contextual information. The Context Definitions provide a uniform structure to the contextual information, which can be stored and shared through the Context Database. Apps query the Context Database to retrieve contextual information which saves the user the effort of entering or finding related information in multiple apps used within the same context.

An implementation of the model on Android is used to demonstrate the Contextual Data Sharing Model. It uses Java classes to represent Context Definitions, which are then instantiated as Java objects and provide a uniform representation of contextual information across apps and devices. The Context Database uses Android's Content Provider interface with SQLite as the storage backend for context entries. The Context Manager acts as a middleware between the apps and the Context Database, and is implemented as a static Java class instantiated in the app's process. The Context Definitions and the Context Manager classes are bundled together into a library which the developers can include in their project to use contexts and interact with the Context Database. Concerns and considerations related to the security permissions and performance of the model are discussed for an Android implementation.

The time required to complete various database operations and its relation to the size of the database is analyzed to identify its impact on performance and usability in the implementation. Conclusions regarding optimization of performance of the queries are also discussed. The impact of running operations on the device was analyzed and found to present no hindrance to the running of other apps on the device.

The demonstration of the movie ticket booking use case shows how apps utilize the Contextual Data Sharing Model to access information which otherwise would have been entered by the user. The resulting user experience reduces user effort and provides relevant information and services through recommendations and suggestions in the app. This allows the user to complete their tasks faster and access relevant information without performing additional steps. The availability of contextual information to apps offers an opportunity to design new features and services that were not previously possible.

The main goal of this research is to enable apps to use the information generated and stored on a device and create contextual services using this information. Apps can present users with services they are most likely require, which saves the efforts related to entering information multiple times across several apps. This leads to better features and an improved user experience due to the availability of contextual information across apps.

10

Future Work

"There's always more information out there."
– Google's 9 Principles of Innovation

The Contextual Data Sharing Model adapts existing approaches and techniques to create an innovative framework which allows apps to create contextual services based on the availability of information. The design and implementation discussed in this thesis is an unique attempt in research of this kind to provide apps with an usable framework utilizing the information already present within apps to create contextual services. The implementation is a proof-of-concept demonstration used to show the impact and viability of the Contextual Data Sharing Model. This research can be further expanded into several areas discussed in the sections below.

Extending to the Cloud

The Context Database used in the implementation uses a SQLite database located on the device. By implementing or extending the Context Database with a database situated in the cloud, it is possible to offload operations which are not possible or feasible on the device. Additional features can be provided on the device by analyzing and mediating contexts between services in the cloud. This allows the contextual information to exist independently from a device, and can be used to share contexts across devices.

Having the Contextual Data Sharing Model extended into the cloud provides access to contextual information from more sources, which can be used to develop more powerful and useful services. This information can be provided to the users in the form of services relevant to their contexts but not present on any particular device. This creates an abstraction between devices and services as the information stored in the cloud can be utilized to provide the same or similar services on any device connected to the ecosystem.

The local datastore such as the one used in this implementation would be a part of the cloud model, and would act as a cache for the Context Database based in the cloud. This helps the performance and QoS of the model on a device as the operations are executed faster with a copy or cache of the datastore located on the device. This also allows apps to work with a limited set of information when access to the cloud is restricted.

Using different Database Software

The SQLite database used in the implementation was chosen due to its lightweight design, stability and availability on a large number of platforms. SQLite lacks several features and capabilities when compared to other database softwares like MySQL [88] and PostgreSQL [89]. This affects the design of the Contextual Data Sharing Model as all operations are based on the nature of queries executed in the Context Database. Different database implementations provide features such as ordering query results and filter results based on parameters, that can be used to return only those results that are contextually relevant. By utilizing other database implementations which provide additional features and services, the features of Contextual Data Sharing Model can also be extended.

One possible and interesting approach would be to use a NoSQL graph database as the storage medium for the Context Database. The graph database can be used to store the various relations and interactions between contexts, which can lead to several new and innovative features. Such an implementation would enable apps to leverage the relational information in the Context Database to design a new generation of contextual services. This approach is similar to data mining where data is analyzed for relations which can be used to provide contextual services.

Adding more Context types

The four context types (*Location, Contact, Event* and *Movie*) discussed in this implementation were developed to provide a proof-of-concept demonstration of the design and working of Context Definitions. Using the Contextual Data Sharing Model on devices requires implementing more use cases as Context Definitions in order to identify and use a significant amount of contextual information. For example, the *Event* context type can be extended to commonly used types such as *Meeting, Movie, Concert, and Lunch*. New context types such as *Restaurant Booking, Article, Task, Project, Conversation, Quote, and Work* that reflect the activities performed by the users using the various apps will allow more relevant contextual information to be identified and utilized.

Storing related services within Context Definitions

The design of Context Definitions and its implementation on Android entails storing the contextual information and its associated methods within a single object. The design can be modified to store services related to that context within the Context Definition itself. The model can be extended to tie in apps with the execution of these services in an abstract way. For example, by including weather and traffic services within an *Event* context, apps that display or use the events can also present this information without querying for weather or traffic data themselves. The system resolves the service request and queries the appropriate app installed on the system to get the required information. This approach allows apps to share services present on the device that can be used to provide related information in more useful ways.

Potential for Wearable devices

The Contextual Data Sharing Model described in this research is designed for smartphone applications. The year 2014 saw the introduction of wearable devices such as smartwatches and fitness trackers, some of which run the same platforms as smartphones. This creates a potential to adapt the Contextual Data Sharing Model for wearable devices.

Wearable devices have comparably lesser capabilities than a smartphone in terms of available memory, processor and screen space. The devices are designed to provide high accessibility to information by being present on the body of the user. Most of these devices require the use of a smartphone which acts as the hub of information which is then sent to the wearable device. The most common use of wearables is to display notifications which the user can interact with. This allows contextual information and services to be displayed to the user when required without requiring them to access a smartphone.

Although information can be displayed and interacted with on a wearable device, the capabilities of this interaction are much less as compared to a smartphone. Actions such as navigation and UI are restricted on a wearable device. This creates a need to show useful and related information so that the user does not have to navigate for the information or access it on the smartphone. This is a situation comparable to the capabilities of a smartphone and a computing device such as a laptop. The lesser capabilities of the smartphone creates a need for contextual services in apps which formed the motivation for developing the Contextual Data Sharing Model. The same motivation can be applied to develop or extend the Contextual Data Sharing Model for wearables.

Connecting with the Internet of Things (IoT)

The Internet of Things (IoT) is the interconnection of uniquely identifiable embedded computing devices within the existing Internet infrastructure. IoT has the potential to offer advanced connectivity of devices, systems, and services that covers a variety of protocols, domains, and applications [90]. The interconnection of these embedded devices, also known as smart objects, is expected to usher in automation in nearly all fields, while also enabling advanced applications like a Smart Grid [91]. According to Gartner, there will be nearly 26 billion devices on the Internet of Things by 2020 [92]. IoT is expected to generate large amounts of data from diverse locations that can be aggregated to create new application areas such as automation and smart homes [93, 94]. This data can be utilized to provide a range of contextual services that allow the user to control and access their IoT devices as resources. Android and iOS have started to integrate IoT functionality by way of APIs that allow users to discover and interact with devices installed in their homes.[1] This creates the potential for integrating the Contextual Data Sharing Model with the IoT devices to provide services that offer users contextual control of their homes. For example, the lights and other IoT devices can be turned off whenever users leave their homes on a trip where they can monitor and control the appliances remotely. The users can also set a preference to turn up the heating in the house based on their expected arrival time.

An iOS implementation

The Contextual Data Sharing Model is designed to be platform agnostic with a bias towards being used on Android or iOS. Even though the implementation is demonstrated using Android, it is possible to implement the model and its components on iOS. The Context Database can utilize the same SQLite version along with the table schema for the backend, while the Context Definitions can be defined using classes in Objective-C or Swift. The challenge in realizing the iOS implementation is the communication between different apps and the Context Database, which cannot be performed using the existing data sharing methods available on the system. The components need to be executed as part of the system to make the API available to all apps on a global level

[1]HomeKit (developed by Apple) is a framework in iOS 8 for communicating and controlling connected accessories in a users home and is integrated with Siri. The Nest Learning Thermostat (developed by Nest Labs, acquired by Google in 2014) is an electronic, programmable, and self-learning Wi-Fi-enabled thermostat available for Android and iOS, which can be controlled using the Google Now application.

and allow exchange of information outside the restrictions of the data sharing methods available to user apps.

Creating a Contextual Ecosystem of Apps, Services and Devices

The Contextual Data Sharing Model can be expanded into several areas connected or associated with a user's context. These include interacting with web services, applications on computing devices such as laptops, smartphones and wearable devices along with interactions with devices not owned by the user such as point of sale devices. Providing interactions and communications between all services the user interacts with allows the possibility of providing context-aware information and features on all these devices. This requires creating and adapting the Contextual Data Sharing Model for each device and platform, and consolidating the information gathered to make it available to services across devices. Such a contextual ecosystem of apps, services and devices will enable the user to interact with tasks on any device and service which creates a continuous user experience based on the user's context.

Bibliography

[1] "Gartner Says Worldwide Traditional PC, Tablet, Ultramobile and Mobile Phone Shipments Are On Pace to Grow 6.9 Percent in 2014." http://www.gartner.com/document/2685317, June 2014.

[2] "MobiLens - Understand Mobile Trends and Consumer Behavior." http://www.comscore.com/Products/Audience-Analytics/MobiLens, Dec. 2014.

[3] S. Keach, "Microsoft says Windows Phone now touts 300,000 apps." http://www.t3.com/news/microsoft-says-windows-phone-now-touts-300000-apps, Aug. 2014.

[4] S. Perez, "iTunes App Store Now Has 1.2 Million Apps, Has Seen 75 Billion Downloads To Date." http://techcrunch.com/2014/06/02/itunes-app-store-now-has-1-2-million-apps-has-seen-75-billion-downloads-to-date/, June 2014.

[5] Phonearena.com, "Android's Google Play beats App Store with over 1 million apps, now officially largest." http://www.phonearena.com/news/Androids-Google-Play-beats-App-Store-with-over-1-million-apps-now-officially-largest_id45680, Sept. 2014.

[6] D. Pogue, "A Place to Put Your Apps." http://www.nytimes.com/2009/11/05/technology/personaltech/05pogue.html?pagewanted=all&_r=0, Sept. 2014.

[7] Americandialect.org, ""App" voted 2010 word of the year by the American Dialect Society (UPDATED) American Dialect Society." http://www.americandialect.org/app-voted-2010-word-of-the-year-by-the-american-dialect-society-updated, Nov. 2014.

[8] S. Perez, "comScore: In U.S. Mobile Market, Samsung, Android Top The Charts; Apps Overtake Web Browsing.." http://techcrunch.com/2012/07/02/comscore-in-u-s-mobile-market-samsung-android-top-the-charts-apps-overtake-web-browsing/, Sept. 2014.

[9] M. Böhmer, B. Hecht, J. Schöning, A. Krüger, and G. Bauer, "Falling Asleep with Angry Birds, Facebook and Kindle: A Large Scale Study on Mobile Application Usage," in *Proceedings of the 13th International Conference on Human Computer Interaction with Mobile Devices and Services*, MobileHCI '11, (New York, NY, USA), pp. 47–56, ACM, 2011.

[10] K. W. Y. Au, Y. F. Zhou, Z. Huang, P. Gill, and D. Lie, "Short Paper: A Look at Smartphone Permission Models," in *Proceedings of the 1st ACM Workshop on Security and Privacy in Smartphones and Mobile Devices*, SPSM '11, (New York, NY, USA), pp. 63–68, ACM, 2011.

[11] B. Chihani, E. Bertin, and N. Crespi, "A comprehensive framework for context-aware communication services," in *Intelligence in Next Generation Networks (ICIN), 2011 15th International Conference on*, pp. 52–57, Oct 2011.

[12] M. Elgan, "Smart apps think (so you dont have to)." http://www.computerworld.com/article/2496110/mobile-apps/smart-apps-think--so-you-don-t-have-to-.html, Mar. 2013.

[13] "Google Now." https://www.google.com/landing/now/, Sept. 2014.

[14] "About Gmail cards." https://support.google.com/websearch/answer/2839480?hl=en, Dec. 2014.

[15] A. Lella, "Top 25 Mobile Apps Dominated By The Largest Digital Media Brands." http://www.comscore.com/Insights/Data-Mine/Top-25-Mobile-Apps-Dominated-By-The-Largest-Digital-Media-Brands, Sept. 2014.

[16] "Sunrise Calendar." https://calendar.sunrise.am/, Sept. 2014.

[17] "x-Callback-Url - iOS Interapp Communication." http://x-callback-url.com/, Sept. 2014.

[18] "Fantastical." https://flexibits.com/fantastical-iphone, Sept. 2014.

[19] B. Schilit, N. Adams, and R. Want, "Context-Aware Computing Applications," in *Proceedings of the 1994 First Workshop on Mobile Computing Systems and Applications*, WMCSA '94, (Washington, DC, USA), pp. 85–90, IEEE Computer Society, 1994.

[20] B. N. Schilit and M. M. Theimer, "Disseminating Active Map Information to Mobile Hosts," *Netwrk. Mag. of Global Internetwkg.*, vol. 8, pp. 22–32, Sept. 1994.

[21] C. Bolchini, C. A. Curino, E. Quintarelli, F. A. Schreiber, and L. Tanca, "A Data-oriented Survey of Context Models," *SIGMOD Rec.*, vol. 36, pp. 19–26, Dec. 2007.

[22] A. K. Dey, "Understanding and Using Context," *Personal Ubiquitous Comput.*, vol. 5, pp. 4–7, Jan. 2001.

[23] A. Zimmermann, A. Lorenz, and R. Oppermann, "An Operational Definition of Context," in *Proceedings of the 6th International and Interdisciplinary Conference on Modeling and Using Context*, CONTEXT'07, (Berlin, Heidelberg), pp. 558–571, Springer-Verlag, 2007.

[24] J. L. Crowley, J. Coutaz, G. Rey, and P. Reignier, "Perceptual Components for Context Aware Computing," in *Proceedings of the 4th International Conference on Ubiquitous Computing*, UbiComp '02, (London, UK, UK), pp. 117–134, Springer-Verlag, 2002.

[25] A. Kofod-petersen and J. Cassens, "Using activity theory to model context awareness," in *Modeling and Retrieval of Context: Second International Workshop, MRC 2005, Revised Selected Papers. Volume 3946 of Lecture Notes in Computer Science*, pp. 1–17, Springer Verlag, 2006.

[26] K. Henricksen and J. Indulska, "Developing Context-aware Pervasive Computing Applications: Models and Approach," *Pervasive Mob. Comput.*, vol. 2, pp. 37–64, Feb. 2006.

[27] A. K. Dey, G. D. Abowd, and D. Salber, "A Conceptual Framework and a Toolkit for Supporting the Rapid Prototyping of Context-aware Applications," *Hum.-Comput. Interact.*, vol. 16, pp. 97–166, Dec. 2001.

[28] G. D. Abowd, A. K. Dey, P. J. Brown, N. Davies, M. Smith, and P. Steggles, "Towards a Better Understanding of Context and Context-Awareness," in *Proceedings of the 1st International Symposium on Handheld and Ubiquitous Computing*, HUC '99, (London, UK, UK), pp. 304–307, Springer-Verlag, 1999.

[29] H. Chen, "An Intelligent Broker for Context-Aware Systems," in *In Adjunct Proceedings of Ubicomp*, pp. 183–184, 2003.

[30] A. Battestini, C. Del Rosso, A. Flanagan, and M. Miettinen, "Creating next generation applications and services for mobile devices: Challenges and opportunities," in *Personal, Indoor and Mobile Radio Communications, 2007. PIMRC 2007. IEEE 18th International Symposium on*, pp. 1–4, Sept 2007.

[31] A. Battestini and J. A. Flanagan, "Analysis and Cluster Based Modelling and Recognition of Context in a Mobile Environment."

[32] N. Malik and U. Mahmud, "Future challenges in context-aware computing," in *Proceedings of the IADIS*, pp. 306–310, 2007.

[33] T. Strang and C. Linnhoff-Popien, "A Context Modeling Survey," in *In: Workshop on Advanced Context Modelling, Reasoning and Management, UbiComp 2004 - The Sixth International Conference on Ubiquitous Computing, Nottingham/England*, 2004.

[34] S. Yau, F. Karim, Y. Wang, B. Wang, and S. K. S. Gupta, "Reconfigurable context-sensitive middleware for pervasive computing," *Pervasive Computing, IEEE*, vol. 1, pp. 33–40, July 2002.

[35] A. Klein, C. Mannweiler, J. Schneider, and H. Schotten, "Access schemes for mobile cloud computing," in *Mobile Data Management (MDM), 2010 Eleventh International Conference on*, pp. 387–392, May 2010.

[36] P. Korpipaa, J. Mantyjarvi, J. Kela, H. Keranen, and E.-J. Malm, "Managing Context Information in Mobile Devices," *IEEE Pervasive Computing*, vol. 2, pp. 42–51, July 2003.

[37] A. Alidin and F. Crestani, "Context acquisition in just-in-time mobile information retrieval," in *Information Retrieval Knowledge Management (CAMP), 2012 International Conference on*, pp. 203–207, March 2012.

[38] D. B. Leake, R. Scherle, J. Budzik, and K. Hammond, "Selecting Task-Relevant Sources for Just-in-Time Retrieval," in *In Proceedings of the AAAI-99 Workshop on Intelligent Information Systems, Menlo Park, CA*, AAAI Press, 1999.

[39] P. Falcarin, M. Valla, J. Yu, C. Licciardi, C. Fr, and L. Lamorte, "Context data management: an architectural framework for context-aware services," *Service Oriented Computing and Applications*, vol. 7, no. 2, pp. 151–168, 2013.

[40] R. Lowe, P. Mandl, and M. Weber, "Context directory: A context-aware service for mobile context-aware computing applications by the example of google android," in *Pervasive Computing and Communications Workshops (PERCOM Workshops), 2012 IEEE International Conference on*, pp. 76–81, March 2012.

[41] B.-G. Chun, S. Ihm, P. Maniatis, M. Naik, and A. Patti, "CloneCloud: Elastic Execution Between Mobile Device and Cloud," in *Proceedings of the Sixth Conference on Computer Systems*, EuroSys '11, (New York, NY, USA), pp. 301–314, ACM, 2011.

[42] E. Cuervo, A. Balasubramanian, D.-k. Cho, A. Wolman, S. Saroiu, R. Chandra, and P. Bahl, "MAUI: Making Smartphones Last Longer with Code Offload," in *Proceedings of the 8th International Conference on Mobile Systems, Applications, and Services*, MobiSys '10, (New York, NY, USA), pp. 49–62, ACM, 2010.

[43] A. Fahim, A. Mtibaa, and K. A. Harras, "Making the Case for Computational Offloading in Mobile Device Clouds," in *Proceedings of the 19th Annual International Conference on Mobile Computing &#38; Networking*, MobiCom '13, (New York, NY, USA), pp. 203–205, ACM, 2013.

[44] N. Fernando, S. W. Loke, and W. Rahayu, "Mobile cloud computing: A survey ," *Future Generation Computer Systems*, vol. 29, no. 1, pp. 84 – 106, 2013. Including Special section: AIRCC-NetCoM 2009 and Special section: Clouds and Service-Oriented Architectures.

[45] K. Kumar and Y.-H. Lu, "Cloud Computing for Mobile Users: Can Offloading Computation Save Energy?," *Computer*, vol. 43, pp. 51–56, Apr. 2010.

[46] J. Sankaranarayanan, H. Hacigumus, and J. Tatemura, "Cosmos: A platform for seamless mobile services in the cloud," in *Mobile Data Management (MDM), 2011 12th IEEE International Conference on*, vol. 1, pp. 303–312, June 2011.

[47] M. O'Sullivan and D. Grigoras, "The cloud personal assistant for providing services to mobile clients," in *Service Oriented System Engineering (SOSE), 2013 IEEE 7th International Symposium on*, pp. 478–485, March 2013.

[48] M. Baldauf and S. Dustdar, "A Survey on Context-aware systems," *INTERNATIONAL JOURNAL OF AD HOC AND UBIQUITOUS COMPUTING*, p. 2004, 2004.

[49] M. J. Pascoe, "Adding Generic Contextual Capabilities to Wearable Computers," in *Proceedings of the 2Nd IEEE International Symposium on Wearable Computers*, ISWC '98, (Washington, DC, USA), pp. 92–, IEEE Computer Society, 1998.

[50] W3C, "Resource Description Framework (RDF) Model and Syntax Specification." `http://www.w3.org/TR/PR-rdf-syntax/`, Jan. 1999.

[51] W3C, "OWL 2 Web Ontology Language Document Overview (Second Edition)." `http://www.w3.org/TR/owl2-overview/`, Dec. 2012.

[52] F. Curbera, M. Duftler, R. Khalaf, W. Nagy, N. Mukhi, and S. Weerawarana, "Unraveling the Web Services Web: An Introduction to SOAP, WSDL, and UDDI," *IEEE Internet Computing*, vol. 6, pp. 86–93, Mar. 2002.

[53] "JSON - JavaScript Object Notation." `http://json.org/`, Sept. 2014.

[54] "Siri." `https://www.apple.com/ios/siri/`, Sept. 2014.

[55] "Cortana." `http://www.windowsphone.com/en-US/how-to/wp8/cortana/meet-cortana`, Sept. 2014.

[56] “data sharing. (n.d.) Computer Desktop Encyclopedia.” http://encyclopedia2.thefreedictionary.com/data+sharing, Dec. 2014.

[57] “Intents and Intent Filters.” https://developer.android.com/guide/components/intents-filters.html, Sept. 2014.

[58] “Data Management in iOS.” https://developer.apple.com/technologies/ios/data-management.html, Sept. 2014.

[59] “RFC 2045 - Multipurpose Internet Mail Extensions (MIME) Part One: Format of Internet Message Bodies.” https://tools.ietf.org/html/rfc2045, Nov. 1996.

[60] “App Manifest.” https://developer.android.com/guide/topics/manifest/manifest-intro.html, Sept. 2014.

[61] “Apple URL Scheme Reference.” https://developer.apple.com/library/ios/featuredarticles/iPhoneURLScheme_Reference/Introduction/Introduction.html, Dec. 2014.

[62] “Common Intents.” https://developer.android.com/guide/components/intents-common.html, Dec. 2014.

[63] “Introducing OneLink - One smart link to rule them ALL.” http://www.onelink.me/, Dec. 2014.

[64] “URX, Relevant and Native Deep Link Mobile Advertising.” http://urx.com/, Dec. 2014.

[65] “App Links Overview.” https://developers.facebook.com/docs/applinks/overview, Dec. 2014.

[66] “AppURL connects native apps to the web with http URLs.” http://appurl.org/, Dec. 2014.

[67] “App Indexing for Google Search - A better search experience for apps and users.” https://developers.google.com/app-indexing/, Dec. 2014.

[68] “Calendar and Reminders Programming Guide.” https://developer.apple.com/library/ios/documentation/DataManagement/Conceptual/EventKitProgGuide/Introduction/Introduction.html, Dec. 2014.

[69] “Address Book Programming Guide for iOS.” https://developer.apple.com/library/ios/documentation/ContactData/Conceptual/AddressBookProgrammingGuideforiPhone/Introduction.html, Dec. 2014.

[70] "Android - Content Provider." https://developer.android.com/guide/topics/providers/content-providers.html, Sept. 2014.

[71] "Storage Access Framework." https://developer.android.com/guide/topics/providers/document-provider.html, Dec. 2014.

[72] "Document Picker Programming Guide." https://developer.apple.com/library/ios/documentation/FileManagement/Conceptual/DocumentPickerProgrammingGuide/Introduction/Introduction.html, Dec. 2014.

[73] "UIPasteboard Class Reference." https://developer.apple.com/library/ios/documentation/UIKit/Reference/UIPasteboard_Class/, Dec. 2014.

[74] "Inter-App Communication." https://developer.apple.com/library/ios/documentation/iPhone/Conceptual/iPhoneOSProgrammingGuide/Inter-AppCommunication/Inter-AppCommunication.html#//apple_ref/doc/uid/TP40007072-CH6-SW2, Dec. 2014.

[75] "Keychain Services Programming Guide." https://developer.apple.com/library/mac/documentation/Security/Conceptual/keychainServConcepts/01introduction/introduction.html#//apple_ref/doc/uid/TP30000897-CH203-TP1, Dec. 2014.

[76] "iCloud for Developers." https://developer.apple.com/icloud/index.html, Dec. 2014.

[77] "BroadcastReceiver." https://developer.android.com/reference/android/content/BroadcastReceiver.html, Dec. 2014.

[78] "Services." https://developer.android.com/guide/components/services.html, Dec. 2014.

[79] B. Venners, "Designing with interfaces : One programmer's struggle to understand the interface." http://www.javaworld.com/article/2076841/core-java/designing-with-interfaces.html, Dec. 1998.

[80] T. Arvin, "Troels' links: Relational database systems." http://troels.arvin.dk/db/rdbms/links/#hierarchical, Sept. 2014.

[81] M. Rouse, "NoSQL (Not Only SQL)." http://searchdatamanagement.techtarget.com/definition/NoSQL-Not-Only-SQL, Oct. 2014.

[82] "SQLite." https://www.sqlite.org/, Sept. 2014.

[83] "Android - Parcelable." https://developer.android.com/reference/android/os/Parcelable.html, Sept. 2014.

[84] Oracle.com, "Lesson 8: Object-Oriented Programming." http://www.oracle.com/technetwork/java/oo-140949.html, Sept. 2014.

[85] A. S. Tanenbaum, *Modern Operating Systems*. Upper Saddle River, NJ, USA: Prentice Hall Press, 3rd ed., 2007.

[86] "System Permissions." https://developer.android.com/guide/topics/security/permissions.html, Dec. 2014.

[87] M. Jovic and M. Hauswirth, "Performance testing of gui applications," in *Software Testing, Verification, and Validation Workshops (ICSTW), 2010 Third International Conference on*, pp. 247–251, April 2010.

[88] "MySQL - The world's most popular open source database." https://www.mysql.com/, Dec. 2014.

[89] "PostgreSQL - The world's most advanced open source database." http://www.postgresql.org/, Dec. 2014.

[90] J. Holler, V. Tsiatsis, C. Mulligan, S. Karnouskos, S. Avesand, and D. Boyle, *From Machine-to-Machine to the Internet of Things: Introduction to a New Age of Intelligence*. Elsevier, 2014.

[91] O. Monnier, "A smarter grid with the Internet of Things, Texas Instruments." http://e2e.ti.com/blogs_/b/smartgrid/archive/2014/05/08/a-smarter-grid-with-the-internet-of-things, Dec. 2014.

[92] Gartner, "Gartner Says the Internet of Things Installed Base Will Grow to 26 Billion Units By 2020." https://www.gartner.com/newsroom/id/2636073, Dec. 2014.

[93] B. Violino, "The 'Internet of things' will mean really, really big data, Inforworld." http://www.infoworld.com/article/2611319/computer-hardware/the--internet-of-things--will-mean-really--really-big-data.html, Dec. 2014.

[94] M. Hogan, "The 'The Internet of Things Database' Data Management Requirements, ScaleDB." http://www.scaledb.com/internet-things-database.php, July 2014.