Ismo Leszczynski

OPC UA App development for Android

Espoo 16.11.2015

Supervisor: Ilkka Seilonen
Instructor: Jouni Aro

Työn lähtökohtana oli selvittää sovelluskehityksen haasteet ja vaatimukset, kun tavoitteena on luoda Android-ympäristöön sen yleisessä ekosysteemissä käytettävää OPC UA -tekniikkaa hyödyntävä sovellus. Selvitystyön pohjalta rakennettiin OPC UA asiakassovellus, joka tarjoaa käyttäjälle OPC UA -spesifikaation määrittelemät toiminnot Androidille ominaisella tavalla.

Author: Ismo Leszczyński  
Title: OPC UA App development for Android  
School: School of Electrical Engineering  
Department: Department of Electrical Engineering and Automation  
Supervisor: Ilkka Seilonen  
Instructor: Jouni Aro  

The number of devices using the Android operating system has skyrocketed in the recent years. At the same time OPC UA applications have become increasingly more popular in industry. Despite of the increased popularity, no significant OPC UA apps for Android have been developed.

The starting point for this thesis was to define the challenges and requirements for developing apps that make use of OPC UA for Android. Based on the findings, an OPC UA app was developed providing users the functionality specified in the OPC UA specification in an Android friendly way.

During the developing process, the suitability of the Android operating system for OPC UA was evaluated and possible use cases were searched. It was concluded that Android offers a flexible platform for development, allowing creation of OPC UA apps. Possible use cases range from maintenance assistance to process control.

Date: 16.11.2015  
Language: English  
Pages: 6 + 51  

Keywords: OPC UA, Android, Java
## Contents

### Tiivistelmä

### Abstract

### Contents

#### List of Acronyms

<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 Introduction</td>
<td>6</td>
</tr>
<tr>
<td>1.1 Background</td>
<td>6</td>
</tr>
<tr>
<td>1.2 Objectives and scope</td>
<td>6</td>
</tr>
<tr>
<td>1.3 Research methods</td>
<td>7</td>
</tr>
<tr>
<td>1.4 Structure of the work</td>
<td>7</td>
</tr>
<tr>
<td>2 OPC Unified Architecture</td>
<td>8</td>
</tr>
<tr>
<td>2.1 OPC and OPC UA</td>
<td>8</td>
</tr>
<tr>
<td>2.2 OPC UA Structure and capabilities</td>
<td>9</td>
</tr>
<tr>
<td>2.2.1 Address Space</td>
<td>9</td>
</tr>
<tr>
<td>2.2.2 Services</td>
<td>12</td>
</tr>
<tr>
<td>3 Android Operating System</td>
<td>13</td>
</tr>
<tr>
<td>3.1 Android apps</td>
<td>13</td>
</tr>
<tr>
<td>3.1.1 App components</td>
<td>14</td>
</tr>
<tr>
<td>3.1.2 Manifest File</td>
<td>16</td>
</tr>
<tr>
<td>3.1.3 Resources</td>
<td>16</td>
</tr>
<tr>
<td>3.2 Android design principles</td>
<td>17</td>
</tr>
<tr>
<td>3.2.1 Enchant Me</td>
<td>17</td>
</tr>
<tr>
<td>3.2.2 Simplify My Life</td>
<td>18</td>
</tr>
<tr>
<td>3.2.3 Make Me Amazing</td>
<td>18</td>
</tr>
<tr>
<td>4 OPC UA app design for Android</td>
<td>19</td>
</tr>
<tr>
<td>4.1 Technical design</td>
<td>19</td>
</tr>
<tr>
<td>4.1.1 Activities vs Fragments</td>
<td>19</td>
</tr>
<tr>
<td>4.1.2 Services</td>
<td>20</td>
</tr>
<tr>
<td>4.1.3 Content providers</td>
<td>21</td>
</tr>
<tr>
<td>4.1.4 Broadcast Receivers</td>
<td>21</td>
</tr>
</tbody>
</table>
4.1.5 User interface ................................................................. 22
4.2 User interface design challenges ........................................... 23

5 Implementation of the OPC UA app ........................................... 25

5.1 Development tools ............................................................. 25
5.2 Structure ............................................................................ 26
5.3 OPC UA client service ......................................................... 26
  5.3.1 Communicating with the service ........................................ 26
  5.3.2 OPC UA connection management .................................... 27
  5.3.3 Implemented OPC UA client features ............................... 28
  5.3.4 Notifications ................................................................. 29
5.4 Activities and Fragments ....................................................... 30
  5.4.1 Main Activity ................................................................. 30
  5.4.2 Connectivity ................................................................. 31
  5.4.3 Connection status ......................................................... 35
  5.4.4 Navigation ................................................................. 36
  5.4.5 Menu ............................................................................ 38
  5.4.6 Browsing ................................................................. 41
  5.4.7 Monitoring ................................................................. 45
  5.4.8 Alarms & Events ......................................................... 48
  5.4.9 History ........................................................................ 50
5.5 Future improvements .......................................................... 53

6 Possible use cases .................................................................. 55

6.1 Maintenance ........................................................................ 55
6.2 Alarm monitoring ............................................................... 55
6.3 Process control ................................................................. 55
6.4 Mobile data retrieval ......................................................... 56

7 Conclusions ......................................................................... 57

References ........................................................................... 58
List of Acronyms

ADT  Android Development Tools
API  Application Programming Interface
APK  Android Package
COM  Component Object Model
DCOM  Distributed Component Object Model
GUID  Globally Unique Identifier
OLE  Object Linking and Embedding
OPC  OLE for Process Control (orig.), Open Platform Communications (current)
OPC AE  OPC Alarms & Events
OPC DA  OPC Data Access
OPC HDA  OPC Historical Data Access
OPC UA  OPC Unified Architecture
OS  Operating System
PC  Personal Computer
SDK  Software Development Kit
SOA  Service-Oriented Architecture
SQL  Structured Query Language
UI  User Interface
URI  Uniform Resource Identifier
VM  Virtual Machine
XML  EXtensible Markup Language
1 Introduction

1.1 Background

In the recent years, the smartphone and tablet PC markets have skyrocketed and more and more people carry around intelligent touchscreen devices in their day to day work. At the same time, the number of OPC UA (Open Platform Communications Unified Architecture) applications and solutions has increased significantly. However, no significant applications for the Android OS (Operating System) have been developed even though Android is the leading operating system on the handheld device market. As handheld devices, Androids are significantly different from the traditional desktop PCs (Personal Computers) typically used for interaction with OPC UA. Therefore, they present unique challenges for the design and implementation of OPC UA client applications. This raises the question if Android devices are suitable for industrial OPC UA applications, and if the vast, existing, device base can be made use of for OPC UA.

1.2 Objectives and scope

The goal of this thesis was to investigate the usability of the Android operating system for OPC UA purposes. The aim was to analyze the possibilities provided by the Android app components in the context of OPC UA client functionality.

In addition, the tasks were to find out how an OPC UA app could be implemented on the Android operating system and create a generic client app. Requirements for the app were to follow the Android design principles and provide the following OPC UA functionality:

- connectivity to any OPC UA server
- address Space navigation
- subscribing to Node values
- writing to Node values
- Alarms & Events interaction
- trend views for data
- historical data/events interaction.
Additional tasks were to evaluate Android development tools and find possible use cases for Android OPC UA client apps in the industry.

1.3 Research methods

The main sources for information in this thesis, besides a literature review, have been the OPC UA specification and the Android development reference websites. The core references used are the OPC UA specifications, the *OPC Unified Architecture* by Damm et al [1] and the Android Developers website [13]. The Android operating system is a constantly evolving ecosystem and therefore printed literature is quickly outdated as new versions of technologies are introduced continuously. Books, such as *Programming Android* by Zigurd Menieks et al [28], *Android Design Patterns* by Greg Nudelman [27] and *Professional Android 4 Application Development* by Reto Meier [29], were examined while researching Android, but the most up-to-date information was obtained from the Android Developer website [13].

Due to the lack of suitable references for OPC UA on Android, most of the questions had to be solved with the expertise gained throughout the execution of this study. In addition, previous work by Otso Palonen [33] was used during the implementation.

Testing of the app during development was in a significant role in solving questions and problems regarding the design of the app. The development focused on providing a smooth experience of OPC UA for users, meaning that the user interface had a major role and had to work flawlessly. Because of this, testing was focused on the user interface and was performed manually by different users within Prosys PMS Ltd. The user feedback was used to improve the design repeatedly during the development process.

1.4 Structure of the work

In this thesis, the basic principles of the OPC UA technology and the Android operating system are introduced in Chapters 2 and 3. The design analysis is presented in Chapter 4 followed by the description of the implementation of the new OPC UA client in Chapter 5, where also some suggestions for future development possibilities are included. The two last chapters cover a few possible use cases in an actual industrial environment (Chapter 6) and the conclusions (Chapter 7).
2 OPC Unified Architecture

This chapter gives a general introduction to the OPC UA specification, its brief history, basic structure and capabilities in the scope required for developing an OPC UA client for Android.

2.1 OPC and OPC UA

OPC UA is the successor to the highly successful OPC Classic specification. OPC is an interoperability standard designed for information exchange between devices and systems from multiple vendors. The acronym OPC has had many meanings over the years. Originally it stood for “OLE for Process Control”, or “Object Linking and Embedding for Process Control”, but in recent years new meanings such as “Open Productivity & Connectivity” and “Open Platform Communications” have been introduced to describe its more recent nature. [1, 2, 3]

OPC Classic was originally developed on top of Microsoft’s COM/DCOM (Distributed Component Object Model) technology, and as such was bound to the Windows operating system. The specification was developed in 1996 by an industrial automation industry task force with the purpose to act as a standard for the exchange of data between different software components made by different manufacturers. The specification defines a standard set of objects, interfaces and methods to be used in process control and manufacturing automation applications to allow interoperability. The specifications provide three separate definitions for accessing process data, alarms and historical data; OPC DA (Data Access), OPC AE (Alarms & Events) and OPC HDA (Historical Data Access). [1, 2, 3]

Even though OPC Classic was a major success, it had a few shortcomings that later prompted the development of its successor, OPC UA. The main problems of the OPC Classic specification included the following shortcomings [1, 2, 3]:

- the use of Microsoft’s DCOM technology, which didn’t allow platform-independent implementations
- the separate definitions of the data models
- poor network handling
- the lack of some security features.
Unlike OPC Classic, OPC UA is a platform-independent specification, which integrates all the functionality of the individual OPC Classic specifications into one extensible framework with service-oriented architecture (SOA). The design goals of OPC UA were

- the functional equivalence with OPC Classic specifications
- platform independence
- better security
- extensibility
- comprehensive information modeling. [1, 2, 3]

2.2 OPC UA Structure and capabilities

This section presents how the OPC UA provides information in the Address Space and how it can be accessed and used with the help of Services.

2.2.1 Address Space

OPC UA presents its information through an Address Space, which is defined as “the collection of information that an OPC UA Server makes visible to its clients” [10]. The Address Space is composed of Nodes, which represent the information available on the server, and their References, which represent the connections between different Nodes; some References are used for building hierarchies within the Address Space, others can define the Nodes type by referencing a type definition within the server [1, 11].

Objects are the base for the Address Space structure and serve as a way to organize its information [1, 11]. Often Objects represent real entities, such as servers and devices, which allows easily interpretable organizing of the Address Space.

Every Address Space has a few common Nodes that are available on all OPC UA servers. These serve as entry points to the Address Space when accessing it, since they are guaranteed to exist on all OPC UA servers.

Each Node in the Address Space belongs to a specific NodeClass. There are eight NodeClasses in total and each Node can have only one NodeClass. The NodeClasses are fixed and no new NodeClasses may be defined. [1, 11]

The three most important NodeClasses are the Object, Variable and Method classes.
**Objects** contain Variables and Methods, and are also able to fire events which can be listened to by clients. Objects are used to structure the Address Space; they don’t contain data other than describing the Node with Attributes such as DisplayName and Description. The values of Objects are exposed using Variables, because Objects themselves don’t have a Value Attribute. Objects can be used to group multiple Variables, Methods, or even other Objects. [1, 11]

**Variable** Nodes represent values. The type of each value is defined for each Variable separately. [1, 11] Variables can be used, for example, to represent the distance from range sensors, the temperature from heat sensors or the state of a valve. In general Variables can be used to expose any data values in the Address Space that aren’t represented by References or the Attributes of a Node.

**Methods** are callable operations that clients can invoke within the Object and get results in return. Methods are always called in the context of an Object, so in practice the Object can be considered the owner of the Method even though OPC UA doesn’t define a clear concept of ownership. Each Method defines a set of input parameters that a client must provide and output parameters that a client should expect in return. [1, 11] For example, an Object representing an actuator could have a Method for starting and stopping the actuation.

In addition to the three main NodeClasses there are four NodeClasses for type definitions of Nodes; ObjectTypes, VariableTypes, ReferenceTypes and DataTypes. [1, 11]

**ObjectTypes** are abstractions of Objects, such as an information model for a specific actuator. ObjectTypes define the structure that all Object instances will have, which in this case means all the child Nodes of said Objects. VariableTypes define Variables the same way as ObjectTypes define Objects. [1, 11]

**ReferenceTypes** define the semantics of References between Nodes; they define what the relationship is between the defined Nodes. [1, 11] Every Reference has a source and target Node in the Address Space. The source and target Nodes are not limited to the server’s own Address Space; the references can point to Nodes in a remote Address Space, also on another server.
**DataTypes** are the basic types in which the data is represented in Nodes, such as integers, strings, bytes etc. [1, 11]. In addition to the basic DataTypes each server can define, if needed, new DataTypes to represent the exposed data.

The last NodeClass is the **Views** class. Views are subsets of the Address Space used to show useful sets of Nodes for certain requirements, instead of browsing the entire Address Space for the required information. [1, 11] For example, it is possible to define a View that shows the configuration-related Nodes in the Address Space, or a View that shows the running state of every pump in a process. Views aren’t limited by anything, allowing building Views suitable for any specific task.

Each of the presented NodeClasses defines a set of Attributes for each Node (Figure 1). The BaseNode in Figure 1 isn’t a real NodeClass in itself; it’s only used to describe the common Attributes which every NodeClass has. [1, 11] The defined Attributes are used to describe Nodes, some Attributes are mandatory while some are optional.

Perhaps the most important Attribute is the NodeId. NodeIds are unique for each Node and are used to identify the Node on the server. The NodeId consists of three parts: an Address Space index, an enumerated identifier type, and the identifier itself. The Address Space index refers to a specific Address Space on the given server. The enumerated identifier type defines the type of the identifier. There are four types of identifiers: (i)
numeric, (ii) string, (iii) GUID (Globally Unique Identifier) or (iv) opaque (a byte table in practice) [1]. ExpandedNodeId is another type of Node identifier in which an Address Space URI (Uniform Resource Identifier) is presented in addition to the Address Space index. The URI is optional when using ExpandedNodeIds, but it overwrites the Address Space index when present.

2.2.2 Services

The Services used for the interaction between OPC UA clients and servers are defined in the fourth part of the OPC UA specification. Services are methods that are used by an OPC UA client to access the data on an OPC UA server. [12] They can be considered the core of OPC UA, since all interactions between OPC UA clients and servers are performed through them. Each Service has parameters that are sent to the server during calls, and the server responds with results, which typically include information whether the call succeeded and what were the results. In the case of failure, the server adds the reason to the response.

Services are categorized into Service Sets based on what they are used for. Each set defines similar services and is used by the client to accomplish specific tasks, such as the View set for browsing the Address Space. The OPC UA Services are listed in Table 1.

Table 1: OPC UA Service Sets and their Services and use cases [1].

<table>
<thead>
<tr>
<th>Use case</th>
<th>Service Set</th>
<th>Services</th>
</tr>
</thead>
<tbody>
<tr>
<td>Find servers</td>
<td>Discovery</td>
<td>FindServers, GetEndpoints, RegisterServer</td>
</tr>
<tr>
<td>Manage the connections between clients and servers</td>
<td>SecureChannel</td>
<td>OpenSecureChannel, CloseSecureChannel</td>
</tr>
<tr>
<td>Modify the structure of the server Address Space</td>
<td>NodeManagement</td>
<td>AddNodes, AddReferences, DeleteNodes, DeleteReferences</td>
</tr>
<tr>
<td>Find information in the Address Space</td>
<td>View</td>
<td>Browse, BrowseNext, TranslateBrowsePathsToNodeIds, RegisterNodes, UnregisterNodes</td>
</tr>
<tr>
<td>Find information in a complex Address Space</td>
<td>Query</td>
<td>QueryFirst, QueryNext</td>
</tr>
<tr>
<td>Read and write data and metadata, access history data</td>
<td>Attribute</td>
<td>Read, HistoryRead, Write, HistoryUpdate</td>
</tr>
<tr>
<td>Calling Methods defined by the server</td>
<td>Method</td>
<td>Call</td>
</tr>
<tr>
<td>Subscribe for data changes and Events</td>
<td>MonitoredItem</td>
<td>CreateMonitoredItems, ModifyMonitoredItems, SetMonitoringMode, SetTriggering, DeleteMonitoredItems</td>
</tr>
<tr>
<td></td>
<td>Subscription</td>
<td>CreateSubscription, ModifySubscription, SetPublishingMode, Publish, Republish, TransferSubscription, DeleteSubscriptions</td>
</tr>
</tbody>
</table>
3 Android Operating System

Android is an open source mobile operating system based on Linux currently being developed by Google. It has been developed especially for touchscreen devices such as smartphones, tablet PCs and wearable devices. In addition, it’s available for TVs, cars, game consoles, digital cameras, regular PCs and numerous embedded devices. It’s estimated that the current market share of Android is roughly 80 % of the mobile operating system market [6], which, in addition to being open source, made it the best choice when choosing the operating system for OPC UA client development.

This chapter provides general information on the principles, design and development of Android apps.

3.1 Android apps

Android apps are written in the Java programming language. The development is done using the Android SDK (Software Development Kit) tools, which compile Java code into an APK (Android package) used to install the app on Android devices. [4]

Apps on Android run in their own security sandboxes with their own unique user IDs. Each of the apps files has its permissions set so that only the user ID of the app is allowed to access them. Each process runs on its own VM (Virtual Machine), in isolation from every other app [4]. By default, this leads to a situation where an app has access only to its own components and nothing else, creating a very secure environment where apps cannot access processes they aren’t permitted to.

When needed, it’s possible to give apps access to other apps or parts of the system services. Apps can request permissions to access device data such as contacts, SMS messages, SD cards, camera, Bluetooth etc. Each of these permissions must be accepted by the user during the app installation.

Since Android is an open source environment, the APK files for apps can be, in practice, acquired anywhere. The most common place to acquire them is the Google Play where developers host and sell their apps to users [32]. Alternatively users may opt to download APKs from third party sites or services.
3.1.1 App components

Android apps are composed of app components. There are four major app components[4]:

- Activities
- Services
- Content providers
- Broadcast receivers.

Each of these components is a different point of entry through which the system can enter the app. Each component has a distinct role with a distinct lifecycle, defining how the component is created and destroyed.

Activities represent a single screen with a user interface [4]. As the name suggests, an Activity could be the representation of a single Activity within an app, such as writing an email, taking a picture, writing a blog post etc. Although there can be many activities in an app that relate to each other, each Activity is independent of the others. Therefore, if permissions are set correctly, apps can launch any Activity from another app. For example, if an app for writing a blog post needs to take a photo, it could launch an Activity of a camera app to take the photo, instead of writing a new Activity for taking photos in the app.

Activities, in themselves, have an additional component worth mentioning. Newer versions of Android have introduced Fragments, which represent a behavior or a portion of an interface in an Activity [9]. It’s possible to combine multiple Fragments in a single Activity to build a multi-pane UI (user interface) or use these Fragments in other Activities. Fragments can also be used to represent a whole user interface in itself, and an Activity can have multiple Fragments that it changes based on what is happening in the app. In some cases one can have a single Activity with multiple Fragments representing different functions instead of creating multiple activities separately for each function. Fragments have their own user interface configurations and background logic. In practice they are separate, independent user interface elements sitting on top of a shared Activity. Fragments are tightly bound to the Activity that launches them and follow the Activity’s life cycle.
Services are components that run in the background to perform long-running tasks and operations. Services in themselves don’t have interfaces; they run in the background separately from any app. Other app components can start a Service and bind to it, in order to interact with it. [4] For example, there can be a Service performing some lengthy calculations in the background while the user performs other actions on the device and, later, retrieves the results with the app by binding to the service and requesting the information.

Content providers manage shared sets of app data. Data can be stored in the file system, a SQLite database, the web, or any other persistent, accessible storage location. [4] With the help of Content Providers, other apps can access and change the shared data, as long as the permissions for the content providers are set to allow it. For example, one Content Provider provides apps with the user’s contact information, allowing apps with the right permissions to access and edit the contacts stored on the device.

Broadcast receivers are components that listen for system-wide broadcast announcements. These broadcasts can originate from the operating system, your own app, or even other apps. [4] Many of the broadcasts originate from the system itself; the system broadcasts information about device events, such as the screen turning off or the battery being low. Most of the time, broadcasts are used to let other components know that something is being performed. For example, the earlier mentioned Service performing the lengthy calculations could send a broadcast that the calculations are finished, and if the app is listening for the broadcast with a receiver, it can trigger the retrieval of the results when the broadcast has been received. This way, there is no need to poll the service about the status of the calculations; it is possible to wait for the calculations to complete and send a broadcast informing about the results.

As mentioned earlier in the Activities, apps can launch any other apps Activities. This is an interesting point in the Android OS and highlights a few key points: Activities are, in fact, standalone parts of the app that don’t require the other Activities for functioning. Unlike traditional programs, there is no single entry point (such as the common main-function in other systems). Another interesting feature, regarding security in the situation where an app launches an Activity from another app, is that the Activity launched from an external app doesn’t run in the same process as the app that launched it remotely; the app that launches the Activity runs in its own private process while the remotely launched
Activity runs in the process that belongs to the app that it’s originally part of [4]. This feature is very effective in protecting the remote-Activity-launching app from process tampering.

3.1.2 Manifest File

One of the most important parts of an app is the Manifest File. The Android system requires information on (i) what the app components are, (ii) what permissions are needed, (iii) what are the requirements that the app has for the API (Application programming interface), (iv) what hardware and software features are required etc. All of this is declared in the app Manifest File. [4]

During the app installation, the Android system goes through the Manifest File and (i) asks for approval for the requested permissions from the user, (ii) checks that the required API level is supported by the device (i.e. that the Android OS version is new enough), (iii) checks if the required hardware and software components are found and (iv) checks if all the requested features are available [4]. The contents of the Manifest File are critical; if the declarations are wrong or insufficient, or requested permissions are insufficient, the app won’t run properly or won’t run at all. Thus, extra caution must be taken when declaring the contents of the Manifest File.

3.1.3 Resources

In addition to the code and components, apps require resources such as images, audio files, and anything that’s related to the visual presentation of the app [4]. App Resources are a way to declare these external assets and reference them in the code. For example, it is possible to declare a certain image as a Resource and give it a symbolic name that can be referenced in the code. If wanting to change the image, it is possible to change the picture in the reference while keeping its symbolic name intact. This allows effective modification of resources without the need to change their references in the code itself.

Another useful and recommended use of resources is the resourcing of text (strings) used in the user interface. When the texts are regarded as a resource and defined in their own file, those texts can be referenced in the code. When wanting to change the texts in the interface, the user can go to the resource and change its contents. This is especially useful for localization; there is no need to change the code. It is enough to change the resource contents to display the right localization. The app may contain multiple text resources for
different languages, and the app can be instructed which resource to load the referenced texts from.

3.2 Android design principles

Android apps don’t have very strict guidelines on how the apps should function. However, there are a few main design principles, provided by the Android User Experience Team that should be taken into account when designing apps for Android. The purpose of these guidelines is to guide app developers to design their apps so that they feel like other Android apps; the whole Android ecosystem benefits from similar design patterns between apps, allowing users to learn the patterns quick and to ease the learning curve when using new apps. In this section the three main Android design principle groups will be introduced [5]:

- Enchant Me
- Simplify My Life
- Make Me Amazing.

3.2.1 Enchant Me

The Enchant Me group proposes ways to unify the design regarding visual effects, sounds, objects and app personalization [5]. The aim is to have all Android apps look and feel alike. The visual themes should be similar, but there may be enhancing effects to improve the user experience. The user should be able to manipulate objects shown in the app directly with touch, to reduce the cognitive effort needed to perform a task, while making it more satisfying. An example of such functionality is the drag and drop functionality in the Android OS, moving things by dragging them with a finger is a natural thought process easy to learn.

One important feature in the Enchant Me group is the ability for apps to learn about their users [5]. Often users enter information in apps, in some form, and the same information is repeatedly needed at later times, when using the app again. Because of this, it’s better for the user experience if the app learns the users’ preferences over time and suggests input options, instead of the users having to enter everything again every time when using the app. An example of this could be the address of a server in a client app; instead of requiring the user to enter the address manually each time, the app could remember recently added addresses and allow the user to pick one of those instead.
3.2.2 Simplify My Life

Simplify My Life is a collection of principles made to help keep things simple in apps. Information should be kept at a minimum, not to overwhelm the user when not needed. [5] At any given time, only the really necessary information should be shown to the user. The use of pictures is strongly encouraged instead of lengthy descriptions, where applicable. When compared with words, pictures are much more efficient in presenting the message to the user.

Emphasis on simple, clear navigation is a key principle in this group. The user should always be able to tell where they are in the app and how they can get from place A to place B. Another important guideline is that features that look the same in different apps should act the same. The pre-defined app elements should never be used in a different way than intended. This makes it easier for users to interact with different apps when the functionality of similar objects is always the same.

3.2.3 Make Me Amazing

The Make Me Amazing principles give guidelines to improve the user experience when interacting with the app [5]. In general, the interaction with apps should be similar. For example, the use of the swipe gesture for navigation is a natural go-to in Android apps, and features like that should be encouraged. The user experience is immensely better when users can figure out ways to interact with the app by themselves without lengthy guides.

An important guideline in this group is how interaction with users should be handled when faced with errors or need for corrections in user input. Feedback to users in such situations should be clear and simple without overly technical details. Users should be able to understand the problem and correct it by themselves without too much trouble [5].
4 OPC UA app design for Android

The focus of this thesis is in the development of the Prosys OPC UA Client app [30]. This section covers the questions regarding the design and choices made during the implementation of the app.

4.1 Technical design

For Android development, it’s important that the designed app indeed acts and feels like an Android app. It’s always possible to simply replicate a desktop PC program to an Android device, but that would defeat the purpose of making an app for touchscreen devices. Since the aim of this thesis is to examine the possibilities of handheld devices, the best practices of Android Developer are followed for the Android app design and development [13].

4.1.1 Activities vs Fragments

When designing an OPU UA app for Android using the Android app components introduced in the previous chapter, it is first necessary to determine how to make use of the components for an OPC UA client. The first major question is how to make use of Activities when implementing the various client functions in OPC UA, such as reading data, listening for events and modifying values. In general, there are two options: either to create an Activity for each client function separately, or to create one master Activity with multiple Fragments for the different client functions. In theory, both options are viable, but during the initial implementation work it was found that the latter is easier to implement in a way that satisfies the requirements for the app developed in this thesis.

The original plan was to consider each OPC UA function as its own Activity. For example, connecting to servers, browsing the server Address Space and reading event history could be considered separate operations and, as such, the separate Activity design seemed like the right option. However, during the first steps of implementing the Activities, it became clear that passing information between Activities was much more complex than passing information between an Activity and its Fragments. The reason for this is that Fragments are tightly bound to their parent Activities, which allows easy communication between them through functions. In case of multiple Activities, there is no straight connection between the Activities that could be used to pass the information. In order to pass information from an Activity to another, it is needed to bundle
information in the call that invokes the other Activity, or use some form of storage to save the information from each Activity [9, 14]. This is much less efficient than having one master Activity handling all the communication between its Fragments, without the need for external storage or calls.

Another reason for favoring one master Activity with multiple Fragments over multiple Activities with single Fragments, in this particular case, is the way states are saved for views. Each Activity and Fragment knows its state only while it’s active [9, 14]. Once an Activity or Fragment is deactivated it loses its state information. This causes the need to save the state if wanting to return to a specific view later and resume its state. This is a very likely scenario in an OPC UA client. For example, users often like to continue browsing the Address Space from the place they left off previously. In the case of multiple Activities it is necessary to save the state information in an external storage, wherefrom it will be loaded when relaunching the Activity. In the case of one master Activity with multiple Fragments, it is possible to save the information to some variable in the master Activity and restore it when the Activity launches the Fragment again.

4.1.2 Services

Services play a major role in the development of an OPC UA client. Since Activities and Fragments are destroyed when they’re deactivated, they aren’t a very good location to run the actual OPC UA client which performs all the connectivity and OPC UA service calls. A better place for the actual client is in a separate Service running in the background, which has its life cycle dictated by the calls received from the client Activity. This allows to isolate the actual client from the user interface effectively, allowing for example to close the app Activity while leaving the client Service running in the background to listen for data changes from the OPC UA server.

Another reason for using a Service for the client is that it is not necessary to wait for the OPC UA service call responses in the Activity itself. Such design allows passing the request to the Service and to keep the Activity running without having to wait for the response, making it possible to keep the user interface responsive even during the long service calls. Once the client Service has handled the service call and received results, it can send a system-wide broadcast with the results that the Activity can receive and act, based on the received results.
The Android Service component offers also useful background workers called AsyncTasks [15]. AsyncTasks run parallel to the thread they’re called from and can be used to run tasks separately. These provide an efficient way to handle OPC UA service calls. Each call can be made into a separate task launched from the client service. This way it is guaranteed that the client service won’t get locked if an OPC UA service call doesn’t respond in a timely fashion.

One more useful feature is the ability to configure the Service to send Android Notifications when pre-defined events happen [16]. For example, using the client it is possible to connect to a server and listen for alarm notifications. Once the connection is established, it is possible to close the Activity and leave the client service running in the background. If the client service receives any alarms, it can be configured to raise Notifications in Android. This way the user can see the OPC UA alarms much like when receiving new emails.

4.1.3 Content providers

While developing a client app, some information, like the recently used connections, app settings or previously received data through OPC UA, should be saved in the device’s internal storage in an easily accessible way.

A good option for storing data is through the SQLite [31] connectivity offered by one of the Android Content Providers [17]. SQLite is a lightweight database solution composed of a single data file and a driver that handles SQL (Structured Query Language) queries to the file. Since SQL is a well-known database technology, it’s a logical choice for the data storage needs. An additional benefit of using SQLite is the possibility of exporting the database from the handheld device if there ever occurs a need to access the collected information on e.g. a desktop PC.

4.1.4 Broadcast Receivers

Broadcast Receivers play a vital role in the communication between different components in an app [18]. When the OPC UA client is implemented, in a separate Service as proposed earlier, Broadcast Receivers must be implemented in the Activity and its Fragments to receive notifications about events from the client Service. For example, the client Service can send a broadcast informing that a certain value subscribed to by the client has changed. This broadcast can be listened for in the Fragment that handles the
display of the data values from the client. When the broadcast is received, the Fragment
can update the UI element to reflect the new value.

4.1.5 User interface

There are three main ways that user interfaces can be created for Android. Interface
layouts can be created in the program logic itself, by using the graphical design tool
provided by the Android development tools, or by manually creating layout files. In most
cases, the best way is to either use the graphical tool or define layouts manually.
Modifications to the user interface from the program logic are best left for situations
where highly dynamic interaction is required.

Layouts are a type of Resource in Android [25]. Layouts resemble the standard XML
(EXtensible Markup Language); each element is a graphical component configured by
special Android-attributes. An example of such an element is shown in Figure 2.

![Figure 2. Android layout element example definition.](image)

In the example above, a TextView element is defined. TextView elements are elements
with text content. In the example, the text, textColor and textSize attributes are assigned
references to other resources defined in the app. Using common resources allows easy
modification of the layout; for example, if a specific textSize resource is assigned to
several elements, only the value of the resource needs to be changed without having to
change the layout code – one change is sufficient to affect all desired text elements.

Both, the graphical design tool and manual layout definition, create identical layout files.
The choice between the methods depends mainly on the developer’s preference.
4.2 User interface design challenges

Perhaps the most challenging part of creating an OPC UA client for Android is the design of the user interface. Traditional desktop PC clients have lots of space to display the information from the server while handheld devices come in all sorts of sizes and can be used both in portrait and landscape viewing mode. When comparing the space difference between a 20” desktop display and a 5” smartphone display, it is understandable that the smartphone implementation must make use of the space available much more efficiently. Aside from the difference in display sizes, an important thing to note is the tool used to interact with the user interface. Mobile devices are usually used with fingers which are much larger and less accurate than a PC mouse or stylus. When used with fingers, for comfortable interaction, relatively large elements are preferable. These differences need to be considered when designing the client user experience on a mobile platform.

Aside from the size difference between traditional PC and handheld device screens, handhelds themselves have significant size differences. The screens for Android devices can be anything between 4” – 10” in most cases. This presents its own unique challenge; the user interface should feel as natural on a 4” device as on a 10” device. [27] The solution to this challenge could be creating customized layouts that are automatically selected based on the device size. An important thing to note is that the experience should feel the same even between different layout options. The users should feel that they’re using the same app, no matter are they using their 4” smartphone or the 10” tablet PC, while at the same time feeling that the app adjusts naturally to the size available on the used device.

Following the design principles introduced earlier, one significant challenge when making an OPC UA client is the amount of information that’s available for OPC UA Nodes. According to the design principles, the amount of data shown should be kept to a minimum not to overwhelm users, and to provide the additional data on request [5]. Keeping this in mind, it’s important to handle all the data available for Nodes in a way that shows the important information up front and provides the additional, less important, details upon request. Variable Nodes are an example of such. The significant information is their Value as they’re primarily used to display values for Objects, and the other bits of information can be hidden from the view, until requested.
Navigation in the app is a challenge of its own. The navigation should be simple, easy to follow and use, but it shouldn’t use too much of the limited display space. A couple of options are the use of swipe gestures to switch between parts of the client or the use of separate navigation pane that hides itself, when not needed, to conserve space on the screen, or even a combination of these two options.
5 Implementation of the OPC UA app

The implementation of the OPC UA app described in this thesis is loosely based on an earlier version of an OPC UA app made by Otso Palonen at Prosys PMS Ltd [33]. Most of the app structure, however, has been created from scratch since the original app wasn’t designed according to the Android app design principles, and it didn’t support all of the features defined in the OPC UA specification. The aim of this thesis was the creation of an Android style app that features nearly all of the OPC UA specifications features. Because of this, only a few of the original apps core tasks related to OPC UA functionality were transferred to the new app.

5.1 Development tools

At the time of implementation, there were two main choices for development tools for making apps on Android: Eclipse with the ADT (Android Development Tools) plugin or the recently released Android Studio [19, 20]. Before Android Studio was released, Eclipse was the primary development environment with the ADT plugin. With the increasing interest in Android, it became apparent that a dedicated development environment was needed to replace the ADT plugin. This brought about the development of Android Studio, a standalone development tool designed specifically for Android apps. Currently, Android Studio is the official IDE (Integrated Development Environment) for Android, although the ADT plugin still receives some updates while people migrate their projects to Android Studio.

For developing the OPC UA client functionality the use of an OPC UA SDK (Software Development Kit) is strongly advised. Since Java is the native programming language of Android, the natural choice is to use a Java SDK for OPC UA, such as the Prosys OPC UA Java SDK [8], which implements the functionality required to connect to OPC UA servers and interact with them with service calls. It is possible to use other languages with tools like Mono [7], but those are outside the scope of this thesis. The use of an SDK allows focusing on the presentation of the data available from an OPC UA server in a form that’s natural to use on a handheld device.
5.2 Structure

Based on the design ideas presented in the previous chapters, the app was divided into two main parts: the OPC UA client Service for OPC UA functionality and the user interaction master Activity with multiple Fragments. In the following sections these components are introduced, and their design and implementation choices are presented, as well as the final functionality from user perspective.

5.3 OPC UA client service

As the target of the implemented app was to provide OPC UA accessibility, the most important part of the app is the OPC UA client that provides the necessary interfaces for OPC UA connectivity and data acquisition functionality. In general, there are two main ways to implement the client: either as a part of the app Activity or as a separate service.

As proposed in section 4.1.2, the most logical solution for this was to have the OPC UA client functionality implemented in a background Service, running separately from the main app. While the client functionality is in a separate Service, the main app design choices won’t affect the operation of the OPC UA client. This leaves room for making design choices in the main app itself, without worrying about the lifecycle of the underlying OPC UA client. Thus, it’s possible to design and implement the app to fulfill the requirements and needs placed by the way information should be presented to the user, without hindering the data acquisition processes.

5.3.1 Communicating with the service

Since the OPC UA client runs separately from the rest of the app, it needs to provide means for communicating between the app and the Service. To accommodate all sorts of communication needs, both synchronous and asynchronous communication must be implemented.

For synchronous communication, such as reading connection information or calling the Service’s functions, Android Services allow apps to bind to them and use the exposed functions as if they were a part of the app itself. This is particularly useful for reading quickly executed functions or passing parameters to function calls directly, without the need to wrap them in separate communication objects.
However, if the service functions take a while to complete, synchronous communication isn’t optimal as it needs to wait for the functions to complete before the app can continue onwards; this can make the app unresponsive during long calls. To remedy this, the Service needs a way to send information back to the app asynchronously without the app having to wait. The best way to implement this is to use Android’s broadcast system introduced in section 4.1.4. Using the broadcast system, the service can send system-wide messages when certain events happen, such as long-running functions complete. The app itself can register to listen for the message types the Service broadcasts and react to them when necessary. This way the app can stay responsive while the Service is processing lengthy function calls. When the service has completed the process, it simply broadcasts the results to the app, and the app can react accordingly.

5.3.2 OPC UA connection management

In the previous work done by Otso Palonen [33], the OPC UA Service could handle only one OPC UA connection at a time. To address this limitation, the current OPC UA app implements additional functionality to manage multiple OPC UA connections within the Service. Whenever functions are called from the app, the app also provides information on which connection functions should be run and the Service routes the function calls to the appropriate OPC UA connections.

Each OPC UA connection is a separate OPC UA client that runs in its own thread parallel to the OPC UA service. This allows running longer OPC UA operations on one connection without affecting the performance of another connection. The workload is divided into separate threads and the master Service thread stays responsive to the requests sent by the app.

In some cases users might want to change settings, such as the connection security. The configuration parameters of the stored OPC UA connections cannot be changed in the current version. Changes require that the user disconnects from the OPC UA server and creates a new connection to it, which erases any subscriptions or event logs that the user has collected for the connection. Future development of the app should include modification of existing connections, allowing the user to change parameters and reconnect automatically while preserving information such as subscriptions after the changes have been made.
5.3.3 Implemented OPC UA client features

Following the OPC UA specification, the developed OPC UA client Service offers basic versions of each major OPC UA client service call. The feature list includes

- browsing the Address Space
- reading Node Attributes
- subscribing to Nodes
- subscribing to Alarm & Condition Events
- reading Node history
- reading Event history
- calling Methods.

To provide smooth interaction with the client Service, each operation is wrapped into separate AsyncTasks. When the app requests a specific operation from the OPC UA client Service, the Service launches the appropriate task with the given parameters. Each task then runs parallel to the service for as long as their operation requires, without causing the client service to get locked. After the tasks are completed they call the completion functions in the main client Service to report results to the app using the broadcast system.

In the current version, only a few variations of these Service calls are used in the OPC UA client Service. In the future work, the variety and configurability of these service calls should be extended to give the user more options to access data in various scenarios and with different server setups. For example, only raw history data can be read for variables, even though in many cases it might be more useful to read the data filtered or modified by the server itself, in which case a different history read function must be used.
5.3.4 Notifications

Since the OPC UA client is running as a separate Service indefinitely, until stopped by the user, it’s important that the user is aware that the service is running. The straightforward way of showing such information is to use Android Notifications, which were presented in 4.1.2.

Notifications can be configured for various uses in various styles. In the app presented in this thesis, the functionality of the Notification is limited to showing the user that the OPC UA client service is running in the background. In addition to showing that the Service is running, the Notification can be expanded to show a Stop button for stopping the Service without having to enter the app itself. The Notification is presented in Figure 3.

![Figure 3. Notification implementation.](image)

Future improvements of the app could include functionality to allow the user to configure the OPC UA client Service to raise custom notifications whenever specified alarms or events happen on the OPC UA server. With this improvement the user would not need to open the app to see if any of the specific alarms or events have occurred.
5.4 Activities and Fragments

This section provides information of the central Activity and Fragments for the various OPC UA client features.

5.4.1 Main Activity

As presented earlier (see 5.2), the OPC UA app consist of a single Activity, referred to as MainActivity from now on, which manages the various Fragments and information storage during runtime. The MainActivity doesn’t have a UI itself as all functionality is performed via the various Fragments containing own UI code within themselves.

The primary functionality of the MainActivity is Fragment management. Every view in the app is a separate Fragment, which is launched through functions embedded in the MainActivity. Whenever transitions happen between the various Fragments, the currently active Fragment tells the MainActivity to change the active Fragment to the desired one.

As pointed out earlier, Activities and Fragments are tightly bound (see 3.1.1), and thus have easy communication possibilities between each other. This is utilized when storing state and configuration information of Fragments, when changing from one Fragment to another. For example, when selecting a Node for history reading purposes in the Browse Fragment (introduced in 0), the Browse Fragment stores the Node information in the MainActivity, after which it tells the MainActivity to launch the appropriate history viewing Fragment. Once the history Fragment has been launched, it requests the Node information from the MainActivity and stores it within itself. This way, communication between Fragments is simple and straightforward and, as an additional benefit, the configuration parameters are stored in the MainActivity and can be reaccessed later without the need to retrieve the information from another Fragment.

One more advantage worth mentioning is that the MainActivity is the gateway between Fragments and an external SQLite database. When the MainActivity is launched, it initializes or connects to an existing SQLite database and provides functions with what the Fragments can access the database. In the current implementation, the information stored in the SQLite database includes the connection history and state information for the MainActivity itself. When the MainActivity closes, it stores its state information in the database and restores it when launched again.
5.4.2 Connectivity

The first step in using the OPC UA client app is connecting to an OPC UA server. For connecting to servers, the first Fragment to be implemented is the connectivity Fragment. The connectivity Fragment must be able to connect to new servers and previously used servers.

Since managing server history and entering details of new server connections requires a considerable amount of space, the connectivity Fragment was split into two separate Fragments – the Connect Fragment and the New Connection Fragment – the second of which would only be invoked by the Connect Fragment.

The Connect Fragment allows the user to either choose to connect to a new or previously used server. The implementation of the Connect Fragment at the time of writing is shown in Figure 4.

![Figure 4. Connect Fragment interface.](image-url)
If the user chooses to connect to a new server, the Connect Fragment invokes the New Connection Fragment. The task of the New Connection Fragment is to allow the user to enter the details for a new server connection, which include the following:

- name for the server
- server URI
- connection security.

In the current version, two security modes are supported. The default security mode is ‘None’, which connects simply without using security at all. The other option is to use Basic128RSA15 encryption which is “a suite of algorithms that uses RSA15 as Key-Wrap-algorithm and 128-Bit for encryption algorithms.” [23] An important thing to note is that not all OPC UA servers provide connectivity with security. Therefore, the users must determine if the server provides such security before they can use the security option. In future implementations, the other security options, specified in the OPC UA Specification, should be implemented and presented as options for the user. Also, certificate support should be added to the UI itself.

The implemented UI for the New Connection Fragment is shown in Figure 5.

![Figure 5. New Connection Fragment interface.](image-url)
Since the New Connection Fragment can be considered a child Fragment of the Connect Fragment, the New Connection Fragment allows for backwards navigation to the Connect Fragment in case the user wishes to choose another connection option. As per the Android back navigation best practices, this is done with the back navigation button on the left side of the app icon located at the top left corner of the screen [22].

Regardless of which connectivity option is chosen for use, the connectivity Fragments use the OPC UA client service running in the background to connect to the desired server. While creating the connection, the app will display an informative loader so that the user knows what’s happening. The loader is shown in Figure 6.

![Figure 6. Connection information loader.](image-url)
If the connection is successful, the user is directed to the Status Fragment, which is presented in the next section. In case of a connection failure, the user is notified about the problem and asked to check the connection details, or try again. The connection failure notification is presented in Figure 7.

Figure 7. Connection failure notification.
5.4.3 Connection status

Once a connection has been established, there should be a feature for showing the connection status at any given time. To fulfill this requirement, the Status Fragment was created. The role of the Status Fragment is to display the information of a selected connection, and provide means to disconnect from the server without closing the entire app. The implemented Status Fragment interface is shown in Figure 8.

![Connection status interface](image)

*Figure 8. Status Fragment interface.*

In its current state, the Status Fragment simply displays the connection details for the active connection and allows the user to disconnect from the server. In the future implementations, the Status Fragment could also feature functionality to modify the connection on the go, allowing the user to make changes without creating a new connection for the same server.
5.4.4 Navigation

After connecting to a server, various OPC UA features become available to the user. To access the different features, a form of navigation had to be chosen and implemented. As one of the aims of this thesis was to develop an app without excessive chaos on the user interface, the natural choice for navigation was a Navigation Drawer. Navigation Drawers are navigation elements that can be brought to view by swiping the side of the device’s screen, or by tapping the navigation drawer icon on the left side of the app icon.

Since Navigation Drawers are – in practice – Fragments that fill only a part of the screen, a Navigation Fragment had to be implemented to provide the desired navigation functionality.

As the app allows simultaneous connectivity to multiple servers, one of the features in the Navigation Fragment has to allow selecting which server is being used at any given time. The navigation should also provide a route to viewing a selected server’s connection details with the Status Fragment. To provide the user with the information on the active connections, the navigation panel should, preferably, display also the connection status information for easier viewing when compared with repeatedly opening the Status Fragment.

In addition to server selection, the navigation must provide means to access the core features of OPC UA i.e. browsing, monitoring and viewing Alarms & Events. Because of these requirements, the Navigation Fragment was implemented as shown in Figure 9.
As shown in Figure 9, the Navigation Fragment consists of two parts: the server selector and navigation buttons. The connection that the navigation buttons utilize depends on which connection is currently selected in the Connections section. Selecting a server on the navigation pane also changes which server the active Fragment in the background is using. For example, if the active background Fragment is the Status Fragment, then changing the server makes the Status Fragment to update the details of the selected server, without having to invoke the Fragment again from the navigation pane. Also, each server has its own connection status information, below its name in the selector, allowing a quick view of the state of the connection.

For future development, the Navigation Fragment should provide means to remove a connection without having to go to the desired connections Status view. For example, the server selection row could have a remove button, on the right edge of the screen, which would allow removal without the extra step of going through the Status Fragment.
5.4.5 Menu

Typically, Android apps contain a generic menu for the app. At the top right corner of the app screen is a menu with a few functions, as shown in Figure 10.

![Android menu](image)

Figure 10. Android menu.

The Stop service option requests the OPC UA client service to terminate by disconnecting all connections and shutting down the app. The Help option opens a dialog with the user manual for the app as shown in Figure 11.
License terms opens the general license terms, which have to be accepted when launching the app for the first time. The License terms view is shown in Figure 12.
Lastly, the About option opens a compact dialog (Figure 13), which includes the app info and a link to the third party libraries, list where users can see which external libraries have been used.

Figure 13. About dialog.
5.4.6 Browsing

To make use of the OPC UA Address Space effectively, the app user needs to be able to browse the Address Space easily and clearly. Even though the Address Space can have circular references, the clearest and easiest-to-understand form of navigating the Address Space, in most cases, is a tree hierarchy. When displayed as a tree hierarchy, navigating the OPC UA servers Address Space is similar to navigating a standard PC file system, and thus very familiar to most users; there is no need of having extensive knowledge of OPC UA itself.

On mobile devices, a tree hierarchy is problematic because of its horizontally expanding nature; the width of the screen is limited and each item in the tree needs to be large enough that the user can see it clearly and select it with reasonable ease. The size requirement for the visualization of the tree hierarchy would require the app to provide two axes, horizontal and vertical, scrolling to see each item in the tree when the tree grows larger, which isn’t intuitive nor easy to use. The problem is visualized in Figure 14.

![Tree visualization problem, tree view from UaExpert](image)

*Figure 14. Tree visualization problem, tree view from UaExpert [26].*

As seen in Figure 14, a tree view can easily expand beyond the display on a mobile device. One might argue that the tree elements could be smaller, but with an extensive tree that won’t solve the underlying issue. Naturally, larger devices provide more space and allow
larger elements to be shown, but the aim of this thesis was to develop an app that works on devices of any size, meaning that also small phones and limited screen area must be taken into account.

To solve the problem caused by the tree navigation, a different form of traversing the tree was implemented. Instead of a single tree view, the navigation was implemented using two components: a breadcrumb trail [21] to display the current position in the hierarchy, and a list for displaying the contents in the current hierarchical Node. This design allows displaying tree hierarchies of any size in a compact view, suitable for mobile devices. Only a single row is needed for displaying the current position in the hierarchy and the remaining space can be used to display the contents of the current Node in a list form. The implemented view is presented in Figure 15.

![Browse view implementation](image)

*Figure 15. Browse view implementation.*

As seen in Figure 15, the breadcrumb navigation component is located at the top of the view and the list of Nodes under the current hierarchical Node is located beneath. The
The Browse Fragment offers two directions for traversing in the Address Space: backwards navigation using the breadcrumb view and forward navigation using the Node listing below the breadcrumbs. Selecting Nodes in the list view navigates the user one step deeper into the tree, if the selected Node has suitable references to other Nodes. Selecting a Node in the breadcrumb view navigates the user backwards in the hierarchy to the selected Node.

The implemented navigation design also addresses a challenge that’s presented by the nature of the OPC UA reference system within the Address Space. As mentioned earlier, circular references are entirely possible and allowed by the OPC UA specification. While presenting circular references in a navigation path selected by the user would be problematic in a traditional tree view as the tree could expand indefinitely, the implemented breadcrumb navigation can display the path entirely. Each time the user navigates deeper into the Address Space the current Node is added to the navigation path seen in the breadcrumb view. The entire navigation path is stored from start to end, so the user can jump back to any point in the navigation history through the breadcrumbs. Such navigation is familiar to most users as it is very similar to typical internet browser backwards navigation or most operating systems’ folder navigation where it is possible to either navigate one step backwards or to select a particular step in the navigation history.

In addition to navigation, the Browse Fragment provides the user with shortcuts to interact with the various Nodes in the Address Space. If a Node in the Node list offers any of the OPC UA services, aside from the navigation functionality, an icon depicting the service feature is displayed on the same row with the Node. The supported features and their icons are shown in Table 2.
Through the feature icons, presented in Table 2, the user gains access to all of the features provided by the app. The Display info button opens a dialog where the Node information is displayed from the server. The Subscribe and Unsubscribe buttons control which Nodes the OPC UA client service subscribes to for value updates. The Data Value history, Event history and Call method buttons invoke their respective Fragments through the MainActivity.

Table 2. Browse Fragment features and their icons.

<table>
<thead>
<tr>
<th>Feature</th>
<th>Icon</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subscribe</td>
<td><img src="image" alt="+" /></td>
</tr>
<tr>
<td>Unsubscribe</td>
<td><img src="image" alt="x" /></td>
</tr>
<tr>
<td>Data Value history</td>
<td><img src="image" alt="history" /></td>
</tr>
<tr>
<td>Event history</td>
<td><img src="image" alt="event" /></td>
</tr>
<tr>
<td>Call method</td>
<td><img src="image" alt="call" /></td>
</tr>
<tr>
<td>Display info</td>
<td><img src="image" alt="info" /></td>
</tr>
</tbody>
</table>
5.4.7 Monitoring

Once Nodes have been subscribed to for monitoring, the user should have some way of seeing what values the selected Nodes have. In most cases it’s useful to see all subscribed Nodes in a single view, without any additional clutter, so a new Fragment had to be created to provide such functionality. In addition to viewing values, users often wish, for various reasons, to change the value of a Node on the server, so the same view should provide means to edit the displayed values on demand.

To provide the required functionality, the Monitor Fragment was created. The Monitor Fragment displays every subscribed Node and its value in a list for easy viewing. The Fragment also allows editing and removing subscribed Nodes. The implemented interface is shown in Figure 16.

![Figure 16. Monitor Fragment interface.](image-url)
As seen in Figure 16, the subscribed Nodes are shown in a list with only the name and value of the Node. To keep the view clutter free, the additional functions for editing and removing the Nodes have been placed on the status bar in the top right corner of the view; the pencil icon represents the edit function and the cross the remove function.

If desired to modify a value the user must select the row and press the edit icon. Once the edit icon has been pressed the Fragment displays a dialogue to enter the new value as shown in Figure 17.

![Figure 17. Edit Node value dialogue.](image)

In addition to monitoring and value editing, the Monitor Fragment features also a quick filter for switching between viewing all subscribed Nodes on all connections and viewing only the active server’s subscriptions. The menu can be found in the top left corner of the view as shown in Figure 18.

![Figure 18. Monitor server selector.](image)
In future, a significant improvement for this view would be the introduction of a graph view for selected Nodes. The user could subscribe to a Node and follow its values in a dynamically self-updating graph with a desired timeframe. Seeing only current values isn’t often useful, it’s more useful to see how the value has changed over time. A static implementation of such a graph is introduced in 5.4.9 when historical data viewing is introduced. A similar implementation could be created for live data as well.
5.4.8 Alarms & Events

In the case that the OPC UA server provides OPC UA events, the client app needs to be able to show them to the user in some form. As these events are, in practice, text messages with additional information bundled to them, the natural way to display them is a scrolling list ordered by time. Typically, event messages also contain a few other important parameters such as severity and source, which should be shown along with the message in the listing. The implemented view is presented in Figure 19.

![Figure 19. Alarms & events view.](image)

As with the Monitor Fragment, presented in 5.4.7, the user can choose to view only the selected server’s event list or view all servers at the same time using the dropdown menu located in the top left corner of the view.

Each alarm and event row highlights the severity of the item with a colored box, in which the severity code is displayed. The purpose of the emphasis is to help the user notice
which events are important based on their severity. For example, critical alarms are displayed in red for high priority and generic status messages are displayed in green for low priority. The severity values have been divided into three color ranges:

- green for low severity range 1–333
- orange for medium severity range 334–666
- red for high severity range 667–1000.

The severity ranges have been selected from the OPC UA specification [24].

Since each alarm and event contains more information than can be displayed on the screen in a sensible way, the user can view more details by selecting the desired row in the event list. Selecting an item on the list opens a dialog with additional information about the item. The dialog is presented in Figure 20.

![Figure 20. Event details dialog](image)

Future improvements to this Fragment could include Alarm acknowledgement functionality and filtering the received events dynamically on the user interface.
5.4.9 History

If an OPC UA server offers historical data for Nodes, the user might be interested in seeing what has happened during a specific interval in the measurements or events. For such functionality the app provides two different data history features: Data Value and Event History Fragments.

The Data Value History Fragment provides the user two options to retrieve the value history for a Node. The user can select a pre-defined time interval from the past to the current time, or the user can specify a date and time interval for the query. After a time interval has been selected, the user can send the request to the OPC UA client service, which performs the request on the server and returns the results. Once the Fragment receives the results, it displays the data value history as a graph (Figure 21) and a time-value list (Figure 22).

![Graph of data value history](image)

*Figure 21. Data value history graph.*
Figure 22. Data value history list.

After the data has been displayed, the user has two options for browsing the data: (i) the user can use two finger gestures for zooming in and out the graph and (ii) scroll the data on the history list.

If the user wishes to see the event history for a specific Node, then the Event History Fragment is launched. The Event History Fragment is similar to the Data Value History Fragment, as both have the same time filtering options. The only differences are which history function is called from the OPC UA client service and which way the data is displayed. Once the data has been received from the client service, it’s displayed in the same way as the Events Fragment shows live events. Additional information is obtained by selecting a historical event, just like in the Events Fragment. The implemented Event History Fragment is shown in Figure 23.
Both Fragments store the information in a similar way. Once historical data is retrieved, it's stored in the OPC UA client service inside the connection object that was used to retrieve the data. Only the most recent historical read result is stored and replaced with each new call. The reason for storing the data in the service instead of, for example, the SQLite database, is purely performance related. The initial implementation of the historical features showed that storing the information in the SQLite database and then reading it there from to the Fragment was significantly slower than transferring data in the runtime memory only. This led to the decision to leave the SQLite out of the process.

In the future work, the user could be given options for data processing. In some cases the user might want, for example, to store the history data and check it later when not connected to the server. A more concrete example will be provided in Chapter 6.

*Figure 23. Event History Fragment implementation.*
5.5 Future development possibilities

Several specific suggestions for future improvements of the OPC UA client app for specific components were presented in the previous sections. In addition, there are still several possibilities for improvements on general level as well.

The current implementation of the client app is highly fixed configuration wise. The user has very limited configuration options regarding the application functionality. The only features that can be configured are the connection parameters, and even those are configurable only upon creation of a new connection – old connections can’t be modified. Additional configuration options could include, for example:

- automatic connectivity when the app is launched, allowing faster access to the OPC UA server
- server configuration persistence, allowing the user to resume previous configurations on subsequent app uses
- data collection configurability options should include data logging into the SQLite database in case such functionality is required by the user; currently all data is stored only in the device memory
- user interface parameter configuration, users could be provided access to customize their views to suit their needs; currently the views are locked to specific sizes and themes.

The user interface configuration would be a significant improvement to the app. Although the app scales itself dynamically, to a certain extent, based on the device size, users might like to change the view according to their own preferences regarding text size and app themes. Providing such functionality would be a major improvement to the user experience.

For catering different size devices better, the app could also provide functionality to show multiple Fragments at the same time. For example, browsing and monitoring could be shown at the same time on larger devices such as tablets or Android PCs, allowing smoother operation of the app.
Lastly, the app could be further improved by providing (i) application level log information, upon request, and (ii) diagnostic information to experienced users. Such functionality would assist users when problems arise during operation.
6 Possible use cases

Once the design problems are solved, well-designed Android OPC UA apps have an impressive amount of possibilities. Rather than possibilities being an issue, the real issue lies in convincing end users that Android devices could bring benefits when used in an industrial environment with OPC UA. Some concepts for how Android apps could be made use of in an OPC UA environment are presented below, but practically any environment could have an appropriate Android app based on how the environment is used, allowing easy access from handheld devices.

6.1 Maintenance

In an environment where devices are connected to an OPC UA server and provide their information in the Address Space, it would be possible to create apps that assist, for example, maintenance crew in their day-to-day chores. Devices could have QR (Quick Response) codes on them that the Android app could scan. After scanning, the device and all the information it exposes, such as diagnostics information, could be shown on the handheld device.

The easy accessibility to the information with handheld devices would have several benefits: the devices wouldn’t need manufacturer specific diagnostics devices, Android handhelds are abundantly available, and the information would be available for other systems as well through the same OPC UA interface.

6.2 Alarm monitoring

A simple and convenient use case would be OPC UA alarm monitoring on a handheld device. Process operators could connect their devices to a process’s OPC UA server and listen for alarm events from the server, even when away from their workstations. This could improve response times to alarms. Once the alarm is resolved the user could acknowledge the alarm without going to a control terminal separately.

6.3 Process control

Process control is traditionally performed on workstations or panel PCs, but it could also be done through an Android app. The process model could be modified to be touch friendly while keeping the look from the traditional process control terminals, allowing
operators to take their handheld devices to the process area and control it while observing the process up close. This could be useful, for example, when an operator notices that something should be changed in a process while away from the control terminal. Operators could then connect to the process with their handheld devices and do the changes remotely.

6.4 Mobile data retrieval

In some cases, data collection devices have to be installed in locations which cannot be reached by standard communications. In such cases it is often required that data has to be retrieved from the field device. One such way could be with an Android device, when the data retriever could move to the area where the data collector resides and connect to it wirelessly. The data collection device could host a lightweight OPC UA server through which the client app can access the information with history service calls. This way the data could be retrieved with the same kind of software that connects to generic OPC UA devices available elsewhere. Hence, there wouldn’t be need for any manufacturer specific applications for the device.
7 Conclusions

The development environment for the Android OS is highly advanced, allowing rapid development while using generally accepted best practices in application design. The development tools, although new, feel mature and provide developers with powerful ways to simplify otherwise tedious development tasks, such as the extremely flexible user interface designer. All in all, the platform provides a good, open environment for mobile app development with its many tools and flexible distribution routes.

In this work, each of the required OPC UA client features were implemented successfully, proving that Android fulfills the requirements for OPC UA communication. The various components provided by Android can be effectively used to manage all of the required functionality, providing a solid base for App development in OPC UA applications.

The device base for Android devices is immense, allowing one app to be run on a plethora of different devices. The continued increase in OPC UA solutions will certainly bring about implementations on Android as well, thanks to its open source nature and wide usage. There aren’t really any limitations on what sort of solutions could be created for OPC UA on Android devices. Practically, anything that has OPC UA could be presented in a handheld-friendly form.

The main problems when developing Android apps for OPC UA are (i) how to represent the exposed information in an intuitive way easy to follow and (ii) how to make use of it. This thesis presents a way to display information on a generic client level, but custom Information Models offered by OPC UA require special attention on how they should be displayed to users. The problem is not only on handheld devices; a general solution does not currently exist for any OPC UA compliant platforms. Possible solutions could be to standardize generic models that can be expanded and define their visualization. However, a more likely scenario in the near future is simply vendor specific visualization implementations for their own custom information models.
References


http://developer.android.com/reference/android/content/ContentProvider.html

http://developer.android.com/reference/android/content/BroadcastReceiver.html


https://prosysopc.com/products/opc-ua-client-for-android/

https://www.sqlite.org/

http://play.google.com/