An Android Library for Self-Checkout

realizing bidirection SOAP web service communication

Yang Wu

06.2014

PUBLIC ABSTRACT

Yang Wu

The Android Dalvik JVM does not offer high-level support for easy implementation of SOAP Web Service server calls as is available with the Java Enterprise Edition. Even the Java Standard Edition's JAXB package for XML processing cannot be used on Android. And existing Open-Source APIs regarding SOAP Web Service support have turned out to have unacceptable architectural flaws [1]. The development of the new "efficientsoa" Open Source library (https://code.google.com/p/efficientsoap/), which eases efficient and systematic XML serialization and deserialization via Recursive Descent and XML Pull Parsing, has been subject of an earlier thesis guided by Wincor Nixdorf.

Wincor Nixdorf was interested in obtaining a complete library for the Android platform which would realize their interface for Self-Checkout using SOAP Web Service calls to a server. That library should relieve application code from getting in touch with SOAP. While, for such a library, the "efficientsoap" project provided a valuable contribution, more had to be implemented: Firstly, a complete set of abstractions for data types and methods from Wincor Nixdorf's Self-Checkout interface with suitable XML serialization and deserialization based on "efficientsoap" was needed. And as Wincor Nixdorf's interface contains method calls in both communication derections, a polling technique had, secondly, to be realized so as to "fetch" the server-to-client directed method calls.

In this thesis, we will discuss how that Android library for Wincor Nixdorf self-checkout interface has been implemented. Substantial parts of the code have been constructed with the help of code generation techniques. Due to the lack of a server application, testing has been done with a suitable test enviroment with SoapUI.

CONTENTS

LIST OF TABLES

LIST OF FIGURES

CHAPTER 1
INTRODUCTION

The traditional market shopping cannot live without the involvement of people: cashiers, guiders or managers. The absolute reliance on human being can work when there is only limited information to be memorized and processed. It is, however, an unrealistic solution for a modern supermarket. With hundreds of commodities and lists of sale information, inventory, customer identity and etc., there is too much to be handled by human.

As a result, modern retailing systems are closely related to point-of-sale (POS) applications.

After the introduction of electronic cash registers (ECR), the prototype of POS was implemented with buttons and local area network (LAN) in 1974. In the early 1900s, POS was developed on Windows and Unix platforms. And with the advent of Cloud Computing in recent years, the POS systems are able to serve as a web service [2]. The most significant difference between cloud-base POS and traditional one is that all the data is stored in a remote server instead of locally.

In retail industry, a POS solution for applications that consumers encounter directly or indirectly is very common at checkout counter. It combines cash register, card reader, weight scale, conveyor belt, signature capturer and etc. to support the end of a shopping procedure.

Based of the existing POS system, Wincor Nixdorf provides a self-checkout solution to increase revenue by streamlining the shopping experience. Queuing while shopping can be a source of irritation for many customers. Self-checkout offers customers the choice of how they would like to finish their shopping trip, to either serve themselves quickly through the self-service channel as in Figure 1.1, which is particularly attractive for those with a small number of items, or by using the traditional checkout lane, for when more assistance is needed. According to Global EPOS Report, the number of self-checkout system in Europes retail businesses is growing rapidly (44% per annum).

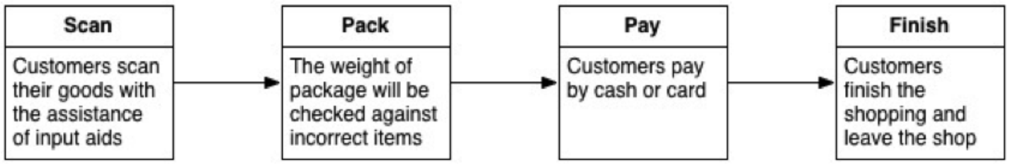

Scan
Customers scan their goods with the assistance of input aids
Pack
The weight of package will be checked against incorrect items
Pay
Customers pay by cash or card
Finish
Customers finish the shopping and leave the shop

Figure 1.1: Self-Checkout System Process.

Additionally, the system increases terminal throughput, boosts productivity and reduces process and personnel costs in an effort to strengthen its competitive position. Employees freed from checkout can be assigned other in-store tasks and thus contribute to greater store productivity.

One problem between POS application and self-service solution is the compatibility. It is possible that the POS server is programmed in C#, while the self-service is developed in Java. Therefore, Wincor Nixdorf provides a market, process and platform independent framework TPiSCAN, as a software solution for the integration of existing POS applications and self-service solution.

There is a POS adapter, which is POS specific. It knows the special properties of the POS and adapts and exposes these properties and features to the self-checkout application. Logically, the self-checkout communicates with the POS adapter. The POS adapter uses a library which is provided by TPiSCAN for the technical communication with self-checkout [3]. TPiSCAN interface has a set of in-bound and out-bound method calls to implement the communication between self-service solution and the POS server of the retail industry.

The TPiSCAN interface provides many benefits. For end-point server application, TPiSCAN is a scalable integration framework, which allows the implementation of a solution for all current and future self-checkout solutions. From development standpoint, the TPiSCAN implements a multi-tier architecture by separating user interface, POS business logic and the access to the peripheral. Both front-end and back-end developers can focus on their own field without worrying about the implementations in the middle.

While more and more customers have already opened their arms to self-checkout service, if the service could be accessed through smart phones, it would be more attractive among the market. Every method call in traditional TPiSCAN interface took almost 3 months to be integrated with POS application. It would save a large sum of efforts if the mobile TPiSCAN could resue the same interface specification. However, the challenge for this transplant is, on the one hand, TPiSCAN uses socket binary connection to communicate with POS application. This is not a good solution for mobile application, as mobile network connection is much less reliable. Thus, the web service takes the place of original communication. However, the new architecture should not loss the clear multi-tier structure. A mobile application will serve as a mobile TPiSCAN endpoint and communicate with POS Server through POS Adapter as in Figure 1.2. The POS Adapter can keep original library, while TPiSCAN mobile application need to be supported by a new library. This thesis will explain how to implements a library for self-checkout with a clear boundary on Android. On the other hand, the two sides of original TPiSCAN burden the similar loads for the communication. In mobile system, the thin client philosophy in mobile devices makes it a better idea to assign more work to the server side. This thesis will also introduce how to use polling technic to realize bi-direction communication logically.

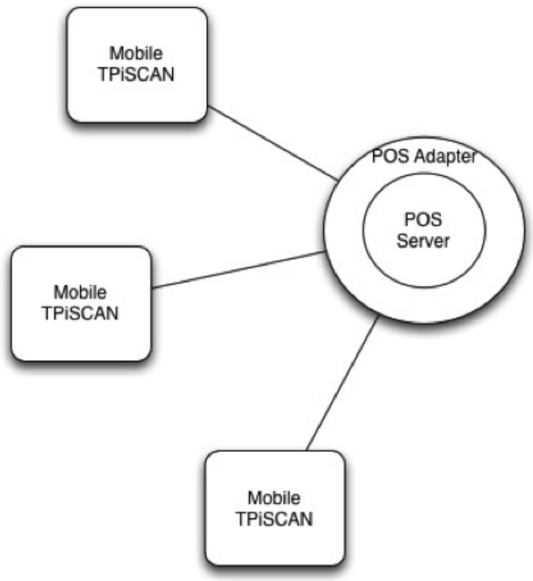


Figure 1.2: Mobile TPiSCAN Architecture.

Before any practical tests can be performed, a server is needed for communication. The implementation of server for mobile self-checkout is a ongoing topic at Wincor Nixdorf. Therefore, I use SoapUI to simulate partial functions of server, a novel and practical way to test the client side when the server is missing.

CHAPTER 2
BACKGROUND

This chapter is divided into four sections. First, I introduce the web service, explaning the reason web service is a proper choice for mobile TPiSCAN application. Second, I provide some background on Android operating system and the characteristic of the platform that makes the implementation challenging. Third, I explain the serialization and deserialization of XML on android platform. Last I discuss the implementation of in-bound methods.

2.1 Web Service

Web Service literally means the service that is available on web. The most important difference between Web Service and Web Site, which is also some resources available online, is Web Site is consumed by human while Web Service is consumed by application.

When do we need web service? The most useful featuren of web service is it enables multiple machines interacting with each other. Comparing with traditional methods, like invoking .jar files remotely, web service has several advantages:

1. The web service can ensure that the result of method call is up to date. With .jar file, every new version application requires a new configuration.
2. It is highly possible that the business logic is implemented with database, which is usually not accessible outside via .jar file.
3. The web service can realize technical-independent, i.e. the application server on Java can still communicate with server on .NET. This is quite significant as the fact that the programing language can hardly be identical on the two sides of networks.

2.1.1 SOAP and REST

There are two main groups: SOAP Web Service and REST Web Service.

“REST (Representational State Transfer) is an architectural style for building client-server applications. SOAP (Simple Object Access Protocol) is a protocol specification for exchanging data between two endpoints.” [4]

SOAP needs to provide protocols itself by WSDL file for service, providing specific response message only after proper input. On the other hand, REST relies on simple URL request, accessing data resource from URL with basic HTTP verbal, like GET, POST, PUT or DELETE.

In another word, REST is “data driven”, while SOAP is “procedure driven” [5] as indicated in Figure 2.1 and Figure 2.11

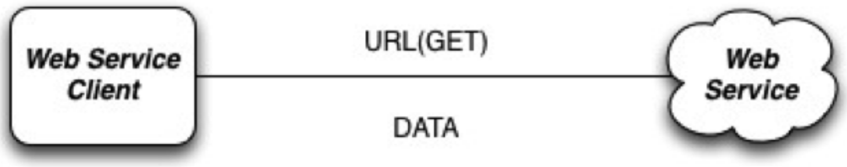


Figure 2.1: REST Web Service.

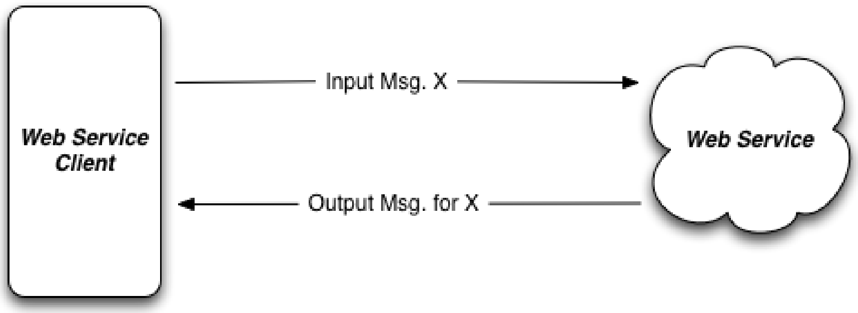


Figure 2.2: SOAP Web Service.

Unlike REST, the data communication in SOAP is always in a message-pair. An input message type will map to a specific output type. In Wincor Nixdorf self-checkout scenario, SOAP is preferred. Because all communications are required to be in a rational sequence.

For example, a request for PIN code has to happen after the use of credit card. With REST, although less workload is needed, it is possible to reverse the sequence of activities.

2.1.2 WSDL

The WSDL, which stands for Web Services Description Language, makes web service technology-independent. To explain WSDL, it is a good idea to compare it with software interface in Figure 2.3.

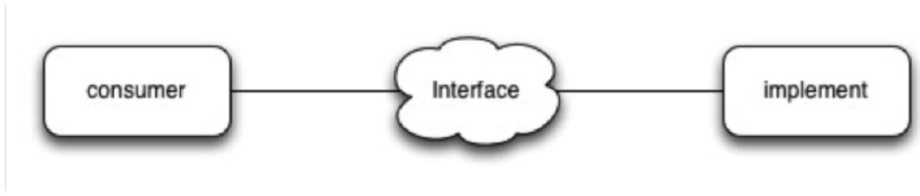


Figure 2.3: Software Interface.

In a typical software interface, the consumer calls the implementation and gets the result via interface, which defines the contract between two parties. The information about input arguments and return type should all be included in the interface.

The relation between WSDL and Web Service is similar as illustrated in Figure 2.4.

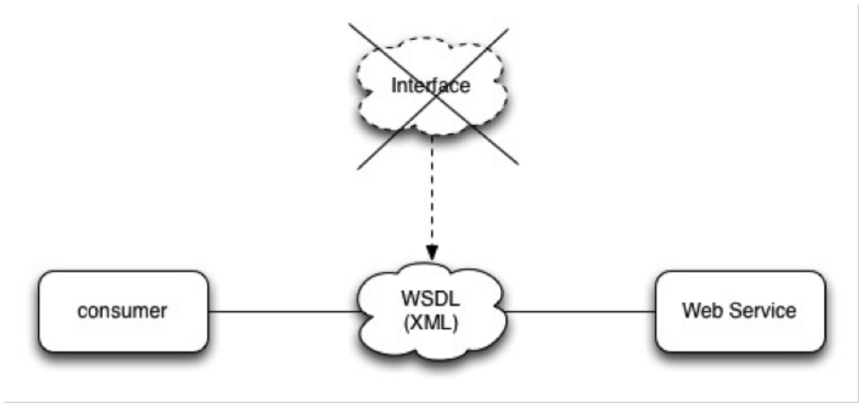


Figure 2.4: WSDL.

As the web service can be implemented in different language, the interface (WSDL) between should also be technique-independent. In traditional software interface, the two parts in different programming language cannot communicate with each other. However, XML file can be parsed and understood by all programing language. That is why XML or WSDL in SOAP is chosen to represent the communication contract. Similar to a software interface, the WSDL defines available methods, input arguments and return type for the communication.

To define a complete WSDL, there are 6 elements needed: <type>, <message> (input), <message> (output), <portType>, <binding> and <service> (see Figure 2.5).

```
<wsdl:definitions ... >
    <types ... />
    <message ... /> [1..2]
    <portType ... />
    <binding>
        <soap:binding... />
        <operation ... />
    </binding>
    <service ... />
</wsdl:definitions>
```

Figure 2.5: Basic Structure for WSDL.

<type> element encloses data type definitions for the communication. <message> element defines the type of input and outpout parameters from schema. If there is only one <message> element, it defines a one-way operation; If there are two <message> elements, it would be a “Request-Response” operation. <portType> provides operations and messages that are involved. <binding> defines message format and protocal details for in <portType>. And <service> groups a set of related ports together.

2.2 Android Background

Android is an operating system based on the Linux kernel. Applications (”apps”), that extend the functionality of devices, are developed primarily in Java programming language with Android software development kit (SDK).

In order to provide better performance to user when there are more than one application running, Android system applies time-shared mechanism to schedule processes and threads. Multiple applications are executed by the CPU by switching among them. But the switches occur so frequently that the user can receives an immediate response. This section will discuss how process and thread work on Android platform, especially how user-interface thread interact with the operating system.

2.2.1 Process

A program loaded into memory and executing is called process. Several processes can be kept in memory simultaneously and share memory via Inter Process Communication.

The process seems to be good enough to accomplish all tasks: it can switch between different tasks in small time slot and use CPU effectively [6]. However, it is not perfect after all. Taking a game program as an example, where graphic processor, audio processor, I/O and so forth will be involved. If there were only one process to handle the program, only one function would be served at one time. The user cannot hear the sounds and see the graphic in the same time. One solution can be creating a new process for an incoming request. However, process creation requires intensive resources. A better solution can be using one process that contains multiple threads.

2.2.2 Thread

A separate thread will be created for a new request. Therefore, as illustrated in Figure 2.6, a normal process contains several threads acting as sub-process of it

A thread of execution becomes the smallest sequence of programmed instructions that can be managed independently by an operating system scheduler for resources.

2.2.3 User Interface Thread

User Interface Thread is a special thread on Android platform, which control the

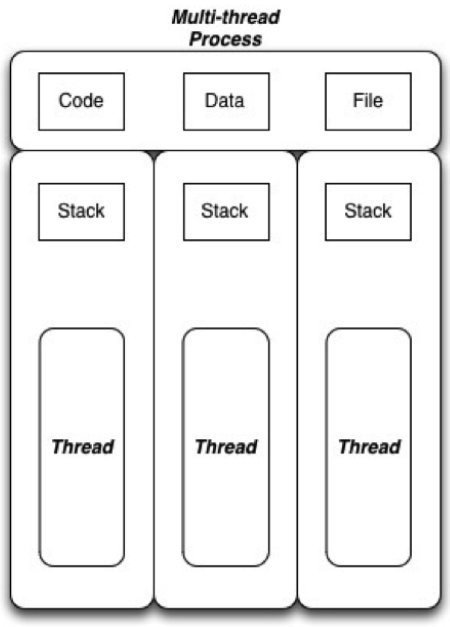


Figure 2.6: A Multi-thread Process.

graphic interactions with the user. It is important because a process typically will executes for only a short time before it either finishes or needs to interact with UI. So UI thread is frequently involved.

"When an application is launched, Android OS creates a thread for application, called main thread. This thread is in charge of dispatching events to appropriate user interface widgets. It is also the thread where application interacts with components form UI toolkit"

This is how Android official website explains the interaction between UI threads (a.k.a., main thread) and mobile operating system. Only one thread is able to access and update the UI toolkit., which is called "single thread model".

One problem with this thread model is the possible poor performance with intensive work in response to user interaction. It is not thread safe. In order not to block the main thread, a many-to-one model in Figure 2.7 is a proper implementation. All other threads are mastered by the main thread. Under a proper mechanism, UI events can be dispatched smoothly.

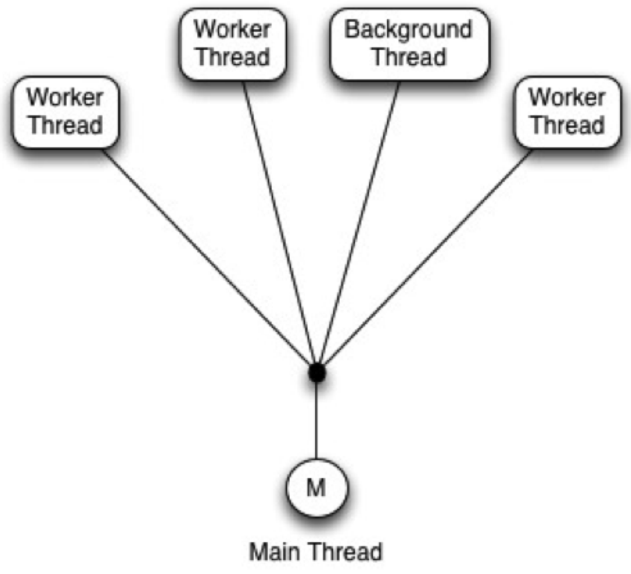


Figure 2.7: Many-to-one Model.

Main thread keeps the control of the whole application, interacting with users. Separated worker threads and background threads execute intensive or long-running workload, like access Internet or query the database. When there is a result for user or an input is required, worker threads or background threads will communicate with the main thread.

On Android, every thread delivers a message to indicate its task. Between this one-to-one mapping of thread and message, Handler object of Android system acts as middleware. It takes care of things like peeking from or adding on the message queue. As in FIgure 2.8, the message queue is exclusively possessed by UI thread. All return results of long-running threads will be queued here and interact with UI screen at different slots according to the scheduling algorithm.

The reason for this separation of the UI thread and worker threads is user interface can only be updated on main thread. If some intensive threads like network connection, XML serialization, parsing were executed at UI thread, the UI thread would be hijacked by these processes and the screen of Android system would freeze or even crash. So, it is safer to put all these long-running operations in background. The UI thread will decide which one to execute when UI update is involved.

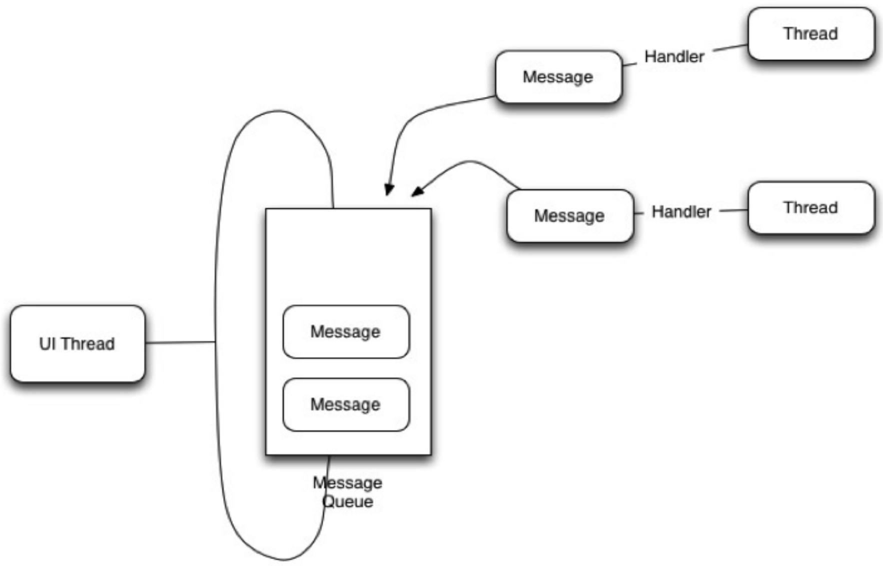


Figure 2.8: Android Message Queue for UI Update.

2.2.4 Worker Thread

One important worker thread that will be used in this thesis is "AsyncTask". "AsyncTask" is an abstract class. It performs asynchronous operations in a worker thread and then publishes the results to the UI thread. To extend AsyncTask class, a subclass must override at least one method doInBackground(Params...), which runs in a pool of background threads. And an "AsyncTask" can be run by calling execute() from the the current thread.

Google IO recommanded to use AsyncTask for long-running tasks. For "thread-safe" reason, Android UI toolkit is only accessible from UI main thread under the single thread model. AsyncTask is such an intelligent thread that takes care of thread management for developers.

2.3 XML Serialize and Deserialize

XML is a popular markup langauge that can model a variety of information concisely. In order to read an XML and understand its contexts, it is necessary for two remote parts to serialize objects into plain XML and deserialize them back. This section will discuss the essential idea about serialization and deserialization used in the library: recursive descent and pull parsing.

2.3.1 Recursive Descent

In regular language, it is impossible to express nested structures. But it is possible in context-free grammar with recursive descnet strategy. The general principle used is the following: consider the recognition of the structure which begins with the start symbol of the target structure as the uppermost goal. If during the persuit of this goal, i.e., while the production is being parsed, an element which is out of current context is encountered, then the recognition of a construct corresponding to it is considered as a subordinate goal to be pursued first, while the higher goal is temporarily suspended [7].

In the implementation, the decision about the steps to be taken is based on the next input. The parser usually looks ahead only by one symbol, as longer 'lookahead' would slow down the process. There is a further example in Figure 2.9 to demonstrate the recursive descent.

```
<Body>
    <Stock>
        <StockName>Facebook</StockName>
        <StockPrice>5.0</StockPrice>
    </Stock>
</Body>
```

Figure 2.9: Example of Recursive Descent of Nested Structures.

In the example, there is the body of a simple nested SOAP body. The uppermost goal here is to recognize the "Stock" element. During the parsing process, the start tags of two other elements ("StockName" and "StockPrice") become the current token. Therefore, the main task for "Stock" is suspended and the subordinate tasks for "StockName" and "StockPrice" gain the priority. After the parser finishes tasks on subordinate layer, it will come back and resume processing the upper layer, i.e., "Stock" element.

The serialization process uses recursive descent strategy in a similar way. The deserialization transforms an XML file into objects by traversing a virtual XML tree, while a serialization process transforms objects into XML by traversing a real object tree.

2.3.2 Pull Parsing

Streaming pull parsing refers to an active programming model. The client application calls methods on an XML parsing library only when it needs to interact with an XML infoset. On the other hand, streaming push parsing refers to a passive programming model. The XML parser sends (pushes) XML data to the application as the parser encounters elements in an XML infoset. Both pull and push parsing are efficient. But the pull parsing has been proved to be a convenient solution for nested data, because it allows the application to use recursive descent.

In the pull parsing model, the client only pulls the XML element when it asks for. Comparing with a popular push paring application, DOM, this strategy provides several advantages. Fist of all, a pull parsing library only reades required elements instead of a entire XML tree. Therefore the library can be smaller and the client code can be simpler. This is very important on mobile platforms, where only limited resources are available. Secondly, the pull parsing strategy can control the parsing process and easily filter elements to be ignored. The API of a pull parsing library can make the deserialization of an XML much more efficient. [8] [9]

Although Android provides XmlPullParsing package to support pull parsing, it is only an abstraction in low-level. The real application would require high-level implementations. The existing open source library “KSoap2-android” is a possible solution. It promises to ease the implementation of SOAP Web Service calls, but has turned out to have unsurmountable flaws especially with complex object structures in the XML-to-Object deserialization. The creation of application domain objects inside the API code is the core design mistake in that Open Source API. [1]

The “efficientsoap” library supports recursive descent XML pull parsing. And as a underlying library, it takes care of SOAP envolope part, leaving the body of SOAP to libraries from upper layers. It is designed in this way because the body of SOAP differs from case to case. Unlike envolope part, “efficientsoap” cannot predict the content of body. However, by providing the methods next() and skipwhitespace(), the “efficientsoap” can traverse the nested structure recursively. And also by providing methods to serialize and deserialize basic types like string or interger, the “efficientsoap” can help in processing complex data types. For example, to serialize the data type in Figure 2.10:

```
public class Car {
    Model model;
    ...
}
```

Figure 2.10: Example of a Complex Data Type.

Class Car has a field model of type Model, which consists of 2 basic elements: Name of string and Year of int. The target of serialization is a SOAP in the format as Figure 2.11.

```
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
    <soap:Header ... />
    <soap:Body>
            <Car>
            <Model>
                <Name>LR3</Name>
                <Year>2014</Year>
            </Model>
            </Car>
    </soap:Body>
</soap:Envelope>
```

Figure 2.11: Example of Target SOAP.

The <Envelope> part will be serialized by “efficientsoap”. As for <Header> and

<Body> parts, the target-specific serializers “CarSerializer” and “ModelSerializer” will extend “EfficientsoapSerializer”, mapping to elements in different layers. As for two elements of basic type: <Name>: string and <Year>: int, they would be serialized via methods from “efficientsoap”.

This thesis will build an Android library based on efficientsoap library.

2.4 In-bound Methods

The communication between client and server happens in two directions. The first one is out-bound, from client to server; the other one is in-bound, from server to client.

The existence of in-bound methods is necessary, as the web service does not only play “callee” of the communication all the time. As an example, if a customer’s signature is required for confirmation after a credit card transaction, the POS application (server) needs to call the client, asking for a signature for confirmation.

In the traditional TPiSCAN interface, the in-bound methods do not have any differences to out-bound ones. Because the two sides are equal, supporting by similar library and providing same interfaces. However, it is not the same case in mobile platform. For one thing, it is hardly possible to implement all the functionality on mobile phones with limited resources. For another, in client-server web service model, it is the client that calls server and starts a communication. The server cannot initiate a method call by itself.

2.4.1 Polling Strategy

Through polling, it is possible to realize in-bound method calls from web service. As mentioned before, web service communications have to be initiated by client. However, the client can trigger a communication which only to sample the status of web service, checking whether there is an in-bound call or not. If there were nothing from server, the client just keeps doing “do-nothing” loop, otherwise the server will start a new serial of method calls.

Polling can fulfill the needs of in-bound method calls logically, however there is still one technical detail should be considered.

After receiving the poll-request, the server can choose different response strategies: either responses only when there is a concrete in-bound method (poll-and-wait) or responses immediately regardless of the content (poll-back).

The first strategy (poll-and-wait) seems to be more efficient. A complete setup for a in-bound method call requires only two communications, which reduces the consumption of bandwidth, processing efforts and complexity. However, while client waiting for a response, the Internet port, worker thread, temporary variable in memory and etc. are all stored or blocked. This problem becomes severe when there is a long period of time that nothing needs to be pushed to client, which is the ordinary case.

Alternatively, "poll-back" strategy focuses on quick responses. Regardless of there is a message to push or not, server will always response a plain text. If there were a method call from server, the response would contain the name of method. This method requires more network resources and process resources of Android system. But considering the fact that polling setup is relative a small procedure, the drawbacks are tolerant comparing with those from "poll-and-wait".

2.4.2 Implementation Logic

After choosing "poll-back" polling as the method to realizing in-bound method calls, this section will discuss the implementation logic in detail.

Every in-bound method has input parameters from server to client. It might be a string of message for explaination or some other information. It is important to decide how to design the return of a poll.

One solution, as in Figure 2.12, is responding a poll request with the name of method ("signature") and input parameters (message). After the client return the value for method "signature()" to server, a in-bound method call from server to client is executed logically.

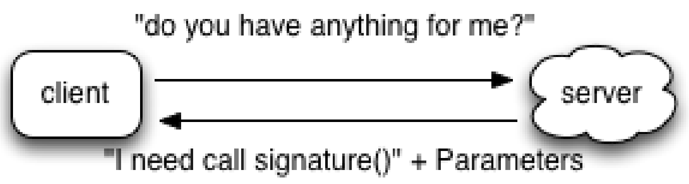

"do you have anything for me?"
client
server
"I need call signature()" + Parameters

Figure 2.12: A Possible Design for Polling Response.

However, the WSDL, which we use to define web service interface, prevents us from doing so. In WSDL, all message types for request and response should predefined one-to-one. As in Figure 2.13, the input for "getSignature" has to be an instance of "getSignatureRequest". And an instance of "getSignatureResponse" can only be returned from the interface if given a proper input.

```
<message name="getSignatureRequest">
    <part name="term" type="xs:string"/>
</message>
<message name="getSignatureResponse">
    <part name="value" type="xs:string"/>
</message>
<portType name="Example">
    <operation name="signature">
        <input message="getSignatureRequest"/>
        <output message="getSignatureResponse"/>
    </operation>
</portType>
...
```

Figure 2.13: Example of WSDL.

In the application, a polling call response needs to carry N differently typed data combination for N possible in-bound parameters. It is not possible to map this variety into the schema for WSDL.

Alternatively, as in Figure 2.14, the response for a polling call can simplly be a string of method name. And the client can fetch in-bound parameters with expected return type in next communication.

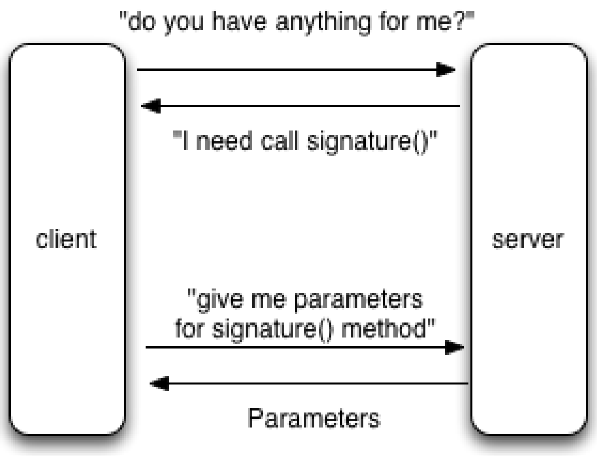

"do you have anything for me?"
"I need call signature()"
client
server
"give me parameters for signature() method"
Parameters

Figure 2.14: A Possible Design for Polling Response.

After having the parameters for an in-bound method, the client can execute the funtion call, return the result and finish the in-bound call. In tradition, a function call from server to client requires only two communications. But affected by web service architecture and WSDL definition, the in-bound method call has to be implemented logically by a “5-way communication” in Figure 2.15:

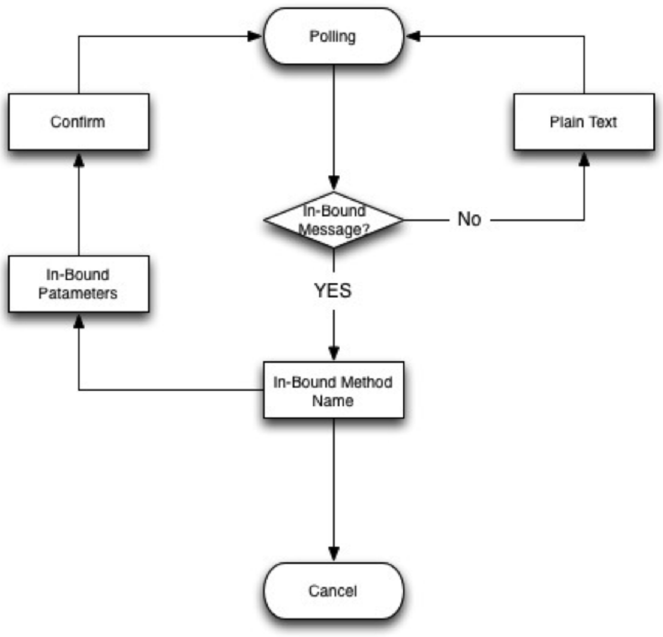


Figure 2.15: Flow Chart for Polling.

1. Client polls the POS application, sampling in-bound method status. (due to web service architecture)

2. Return method name if existed.

3. Client requests for in-bound parameters. (due to WSDL definition)

4. Return required parameters.

5. Client sends back the output of in-bound method.

CHAPTER 3
REQUIREMENT

The purpose of this chapter is to present a description of “Android library for self-checkout realizing bi-direction Soap web service communication”. It will explain the purpose and features of the library, how the libary will behave and the constraints under which it is tested.

3.1 Scope

Android library for self-checkout realizing bi-direction Soap web service communication is an Android library which helps Android applications to serve as a self-checkout application.

The retailing owners can provide customer their own Android application based of the library. The library takes care of things for communication with POS application or POS adapter: serialization, deserialization and Http connection.

Further more, the library should realize communication in two directions. The Android application can not only retrieve information from web service, but it should also be able to provide data to web service.

The following Figure 3.1 sketches this architecture.

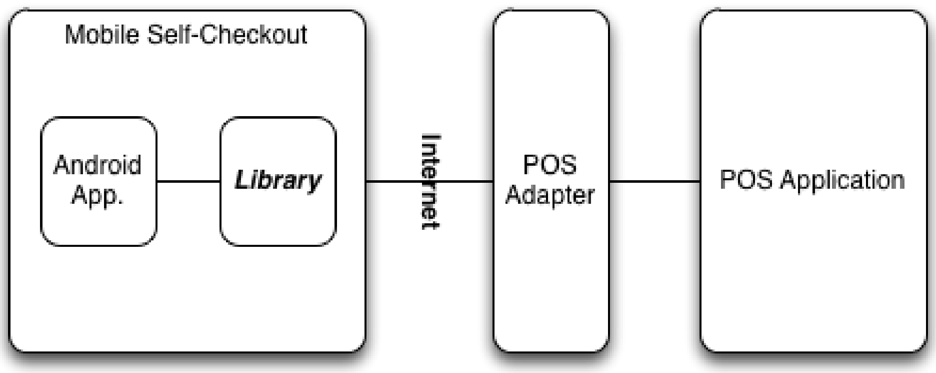


Figure 3.1: Illustration for Mobile Self-Checkout.

3.2 Overall Description

This section will give an overview of the whole library. The library will be explained in its context to show how it interacts with other systems and introduce the basic functionality of it.

3.2.1 Product Perspective

The library needs to communicate with the POS application with the help of "efficientsoap" library. The POS application provides library a bunch of functions (out-bound), but it will also call the service on the library (in-bound), see Figure 3.2.

The library uses "efficientsoap" to deal with underlying technical details. So that the library itself can focus on the compatibility with the existing POS application, which has been integrated with TPiSCAN interface.

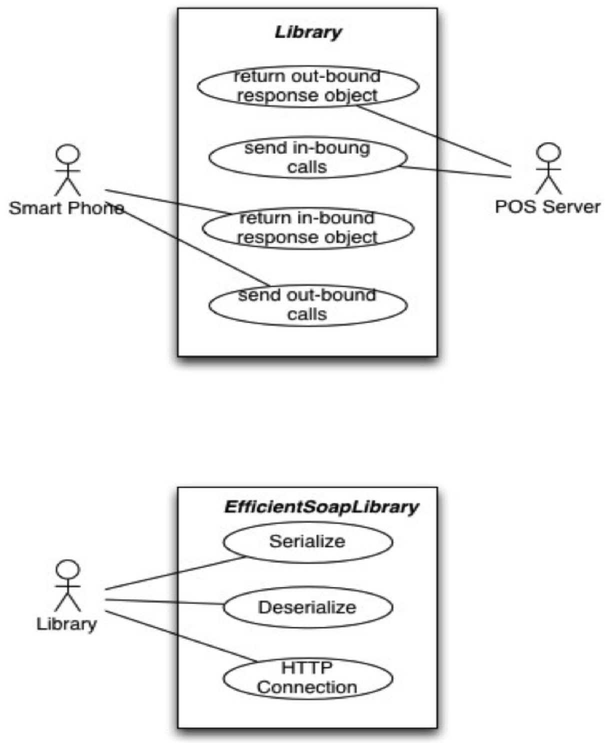


Figure 3.2: Use Case Diagram.

The user and the library should be clear seperated, providing only one entry interface for communications. So that all the technical details will be abstracted for users.

3.2.2 Product Functions

With the library, an Android application can serve as an mobile self-checkout. It should also be able to interact with POS application in the same way as specified in Wincor Nixdorf's TPiSCAN interface.

The App. makes a out-bound function call. The library will help it serialize data objects into XML, and deserialize XML into data objects when the result comes back. For in-bound ones, the library will first deserialize the input parameters for App., and then serialize the return values.

3.2.3 Constrains

First of all, the mobile library is based on unstable network connection. As a result, the traditional TPiSCAN interface with socket connection cannot be transplanted into mobile platform directly.

In addition, the server side is still in a very provisional state. Topic for server will be covered in Wincor Nixdorf at Q3/2014.Therefore there is not a real web serive that can be used for test. A testing enviroment is needed.

3.3 Specific Requirements

This section contains all the functional and quality requirements of the library. It gives a description of the library and its features.

3.3.1 External Interface Requirements

This section describes the software and communication interfaces.

Software Interface

The library provides Android application interfaces for all out-bound web service methods from client to server. In the meanwhile, the Android application should offer implementation for in-bound methods for the library.

Between the library and Android applications, there should be one adapter for the communication. All function calls should through the adapter object.

Communications Interface

The communication between the library and POS application is important. However, in what way the communication is achieved is not important for the library and is therefore handled by the underlying Http proxy from "efficientsoap".

3.3.2 Functional Requirements

The motivation to develope this library is to replace traditional TPiSCAN interface on Android platform, so that customers can go shopping with their smart phones. Therefore, the functional requirements for the library should be consistent with the traditional TPiSCAN interface from Wincor Nixdorf.

A detailed specitication for 66 methods can refer to TPiSCAN Interface Specification from Wincor Nixdorf [3].

CHAPTER 4
EXPERIMENTS

This section has five sections concerning the implementaion. Mock service, serialization-deserialization, thread communication, in-bound polling and library structure will be discussed in order.

All experiments were conducted on Android Emulator. To start with, I need to introduce some information about Wincor Nixdorf self-checkout interface.

There are 4 groups of methods: mainGroup, cancelGroup, posEventsGroup and cancelPosEventsGroup. mainGroup and cancelGroup are two out-bound groups, including methods from client to POS application, while posEventsGroup and cancelPosEventsGroup are in-bound groups.

Each group has its own methods. And each method has its input arguments and output data. Both input and output have a combination of data type in specification. Data types like Integer, Double, Boolean and String are considered to be base types. And there is also a bunch of "composed" types or even "collection" types.

Take "AuthenticationData" as an example, it contains information about the attendant responsibl for a call from the self-checkout to the POS application (out-bound mainGroup).

Table 4.1: AuthenticationData Data Type from TPiSCAN.

Name	**Data Type**	**Remark**
user	String	the attendant's user name
password	String	the attendant's password

As in Table 4.1, AuthenticationData is a nested composed data type, which consists of 2 base type elements "user" and "password".

Among 49 mainGroup methods and 15 posEventsGroup methods in TPiSCAN interface, I choose 2 representative methdos from each group: an out-bound method "getItemDetails" and an in-bound method "signatureRequired" (see Table 4.2 and Table 4.3) for demonstration.

Table 4.2: getItemDetails (out-bound) Method from TPiSCAN.

Name	**Data Type**	**Kind**	**Remark**
authenticationData	AuthenticationData	in	indentify the attendant that controls the call
info	ItemDetailsInfo	in	data that identifies the item for which details are to be reported
itemDetails	ItemDetails	out	detailed information about the item
result	Result	out	the result of the method call

Table 4.3: signatureRequired (in-bound) Method from TPiSCAN.

Name	**Data Type**	**Kind**	**Remark**
message	Message	in	an optional message for explanation
result	Result	out	the result of the method call

An Android library for self-checkout called AppIfcLibrary, which realizes bidirectional SOAP web service communication, is used for demonstration. The purpose of AppIfcLibrary is to transplant traditional TPiSCAN interface into mobile platform and interact with existing POS application through POS Adapter (see Figure 4.1).

In order to accommodate mobile application better, a "session id" data type is added as an argument for every methods from client to server. Because in real application, there would be hundreds of smart phones communicating with the POS application. The session id field can help the server to distinguish one from another.

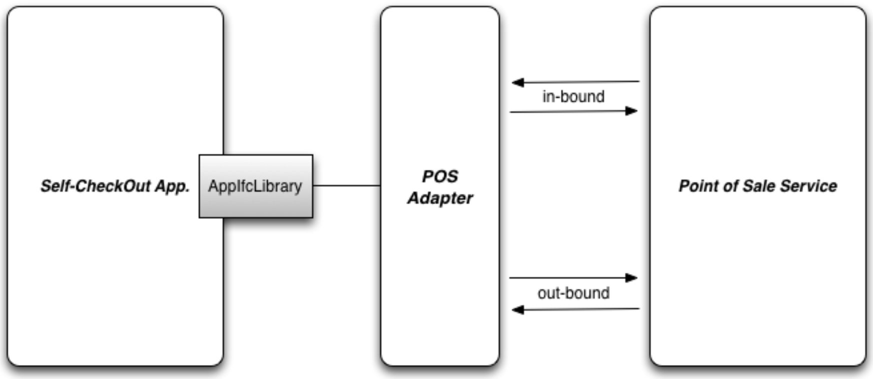


Figure 4.1: AppIfcLibrary and Existing POS.

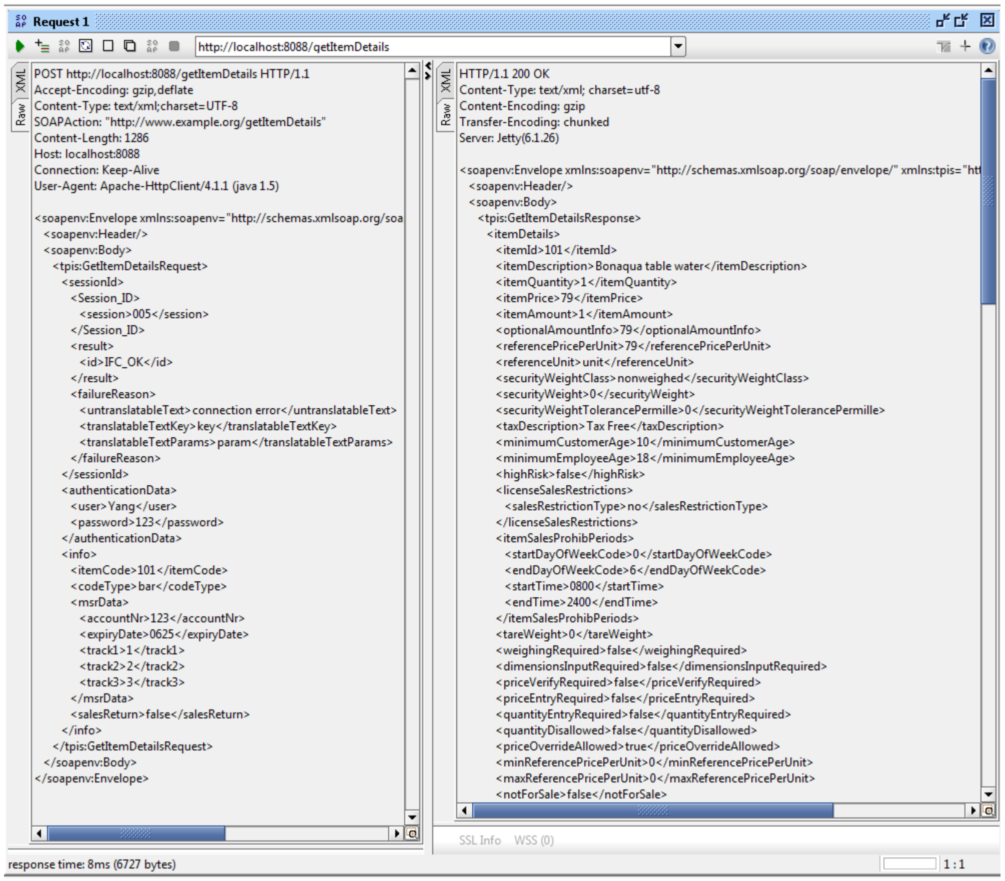
4.1 Mock Service

SOAP, comparing with REST, is more systematic. However, one important factor that prevents SOAP become more popular is the complexity of its WSDL file. This section discusses how to use SoapUI to test client-side application and how to systematically generate WSDL by Wincor Nixdord's code generator for self-checkout interface.

4.1.1 SoapUI

SoapUI is an open-source web service testing application for service-oriented architectures (SOA) and representational state transfers (REST). Its functionality covers web service inspection, invoking, development, simulation and mocking, functional testing, load and compliance testing.

SoapUI is usually used as a test tool for server side application, writing service tests before a test driven development. In the meanwhile, SoapUI also provides a function called "service mocking". It will open a port on local enviroment and response to an incoming message based on the definitions in WSDL.



```
http://localhost:8088/getItemDetails
```

```
POST http://localhost:8088/getItemDetails HTTP/1.1
Accept-Encoding: gzip,deflate
Content-Type: text/xml;charset=UTF-8
SOAPAction: "http://www.example.org/getItemDetails"
Content-Length: 1286
Host: localhost:8088
Connection: Keep-Alive
User-Agent: Apache-HttpClient/4.1.1 (java 1.5)

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soa
  <soapenv:Header/>
  <soapenv:Body>
    <tpis:GetItemDetailsRequest>
      <sessionId>
        <Session_ID>
          <session>005</session>
        </Session_ID>
        <result>
          <id>IFC_OK</id>
        </result>
        <failureReason>
          <untranslatableText>connection error</untranslatableText>
          <translatableTextKey>key</translatableTextKey>
          <translatableTextParams>param</translatableTextParams>
        </failureReason>
      </sessionId>
      <authenticationData>
        <user>Yang</user>
        <password>123</password>
      </authenticationData>
      <info>
        <itemCode>101</itemCode>
        <codeType>bar</codeType>
        <msrData>
          <accountNr>123</accountNr>
          <expiryDate>0625</expiryDate>
          <track1>1</track1>
          <track2>2</track2>
          <track3>3</track3>
        </msrData>
        <salesReturn>false</salesReturn>
      </info>
    </tpis:GetItemDetailsRequest>
  </soapenv:Body>
</soapenv:Envelope>
```

```
HTTP/1.1 200 OK
Content-Type: text/xml; charset=utf-8
Content-Encoding: gzip
Transfer-Encoding: chunked
Server: Jetty(6.1.26)

<soapenv:Envelope xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/" xmlns:tpis="ht
  <soapenv:Header/>
  <soapenv:Body>
    <tpis:GetItemDetailsResponse>
      <itemDetails>
        <itemId>101</itemId>
        <itemDescription>Bonaqua table water</itemDescription>
        <itemQuantity>1</itemQuantity>
        <itemPrice>79</itemPrice>
        <itemAmount>1</itemAmount>
        <optionalAmountInfo>79</optionalAmountInfo>
        <referencePricePerUnit>79</referencePricePerUnit>
        <referenceUnit>unit</referenceUnit>
        <securityWeightClass>nonweighed</securityWeightClass>
        <securityWeight>0</securityWeight>
        <securityWeightTolerancePermille>0</securityWeightTolerancePermille>
        <taxDescription>Tax Free</taxDescription>
        <minimumCustomerAge>10</minimumCustomerAge>
        <minimumEmployeeAge>18</minimumEmployeeAge>
        <highRisk>false</highRisk>
        <licenseSalesRestrictions>
          <salesRestrictionType>no</salesRestrictionType>
        </licenseSalesRestrictions>
        <itemSalesProhibPeriods>
          <startDayOfWeekCode>0</startDayOfWeekCode>
          <endDayOfWeekCode>6</endDayOfWeekCode>
          <startTime>0800</startTime>
          <endTime>2400</endTime>
        </itemSalesProhibPeriods>
        <tareWeight>0</tareWeight>
        <weighingRequired>false</weighingRequired>
        <dimensionsInputRequired>false</dimensionsInputRequired>
        <priceVerifyRequired>false</priceVerifyRequired>
        <priceEntryRequired>false</priceEntryRequired>
        <quantityEntryRequired>false</quantityEntryRequired>
        <quantityDisallowed>false</quantityDisallowed>
        <priceOverrideAllowed>true</priceOverrideAllowed>
        <minReferencePricePerUnit>0</minReferencePricePerUnit>
        <maxReferencePricePerUnit>0</maxReferencePricePerUnit>
        <notForSale>false</notForSale>
```

```
response time: 8ms (6727 bytes)
```

Figure 4.2: Screenshot for SoapUI Mock Service.

As in Figure 4.2, SoapUI open port 8088 for service getItemDetials. And the mock service will simulate Http header and send back response message according to the WSDL. It is undeniable that SoapUI is not intelligent enough. It cannot response differently depending on different parameter values, but only on method names. But it is good enough for testing.

4.1.2 Generated WSDL

The most important thing for mock service is the WSDL file. However, with the help of modified Wincor Nixdorf's code generator, it is possible to generate WSDL files systematically with TPiSCAN interface specification. The code generator first reads a description of the interface methods and data structures into a data model and then executes a "program" consisting of template files. The evaluation of commands and macros in templates files can refer to this data model. There is an example from interface specification for “getItemDetails” in Table 4.4.

Table 4.4: METHOD getItemDetails OF INBOUND GROUP main.

PARAMS	**Data Type**	**Kind**
authenticationData	AuthenticationData	in
info	ItemDetailsInfo	in
itemDetails	ItemDetails	out
result	Result	out

The specification for code generator is simply a mapping of the interface. As for the code generator program, all the command lines starts with a %... keyword and all marcos start with a $... keyword. There is an example of how to generate WSDL file in Figure 4.3.

%foreach ... %endforeach is used for iteration through a collection. “MethodDict.getMethods” returns all the methods defined in the interface specification, including our target “getItemDetails”. %if... (%else...) %endif is used for branch selection. And in the .xsd section, there is a “getXsdName” method, mapping ‘int’ to ‘integer’, ‘String’ to ‘string’, ‘Amoung’ to ‘long’ and etc., specifically for XML Schema Definition syntax [10].

The command in the example first create a .xsd file, defining every method one request-type and one response-type. For getItemdetials method, request-type contains authenticationData:AuthenticationData and info:ItemDetailsInfo and response-type contains result:Result and itemDetials:ItenDetails. Every parameters can be referred to another types in schema file. Then, the command generates the .wsdl file, defining <types>, <message>, <portType>, <binding> and <service> for SOAP communication.

The code generator for WSDL file not only makes the use of SOAP more easy, but also improve the scalability of the project. For one thing, it is tedious to write long WSDL file by hand or create it from J2EE enviroment. For another, the changes in interface can be updated by simply running the code generator.

```
<?xml version="1.0" encoding="UTF-8"?>
<schema ...>
%foreach Method m MethodDict.getMethods
    <element name="$(m.getName:FIRST_UC)Request" type="tns:$(m.getName:FIRST_UC)RequestType" />
        <complexType name="$(m.getName:FIRST_UC)RequestType">
            <sequence>
            %foreach Parameter p m.getInputParameters
                %if p.getType.isSimple
                <element name="$(p.getName)" type="$(p.getType.getXsdName)" />
                %else
                <element name="$(p.getName)" type="tns:$(p.getType.getName:FIRST_UC)" />
                %endif
                 ...
            %endforeach
         ...
%foreach Method m MethodDict.getMethods
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions ... >
    <wsdl:types>
        //... xsd: schema
    </wsdl:types>
    <wsdl:message name="$(m.getName:FIRST_UC)RequestMsg">
        <wsdl:part name="$(m.getName:FIRST_LC)RequestMsg" element="xsd1:$(m.getName:FIRST_UC)Request" />
    </wsdl:message>
    <wsdl:message name="$(m.getName:FIRST_UC)ResponseMsg" ... />
    <wsdl:portType name="TPiSCAN">
        <wsdl:operation name="$(m.getName:FIRST_LC)">
            <wsdl:input message="tns:$(m.getName:FIRST_UC)RequestMsg" />
            <wsdl:output message="tns:$(m.getName:FIRST_UC)ResponseMsg" />
        </wsdl:operation>
    </wsdl:portType>
    ... binding and service
</wsdl:definitions>
%endforeach
```

Figure 4.3: Command to Generate WSDL File.

4.2 Serialization and Deserialization

The serialization/deseriazation processes is the backbone of the library. This section introduces how AppIfcLibrary interacts with efficientsoap library and how recursive descent strategy is implemented.

4.2.1 AppIfcLibrary and Efficientsoap

The “efficientsoap” was designed to be an underlying library that can be used to process SOAP for serialization, deserialization in any case. It only uses methods from Android Operating System and XmlPullParser/XmlSerializer package.

“efficientsoap” is considered to be a general processor for SOAP, any specific application that wants to use it has to construct some higher-level abstractions above. The hierarchy structure can be illustrated by Figure 4.4: only when the application needs to deal with serialization, deserializaiton and HTTP connection, the “efficientsoap”will be called.

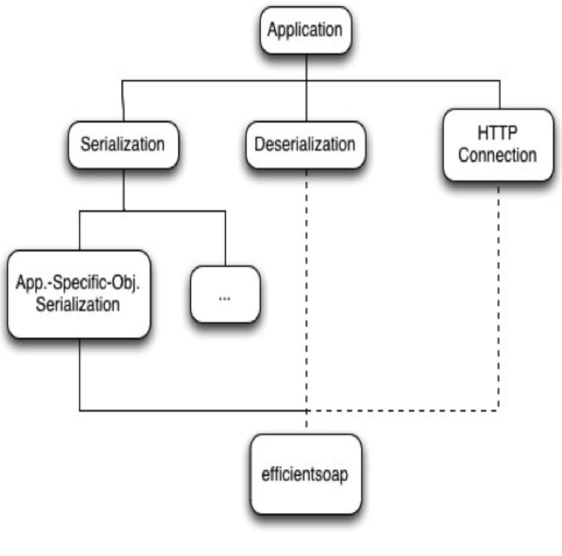


Figure 4.4: How to Use efficientsoap.

The higher-level abstraction is needed here to handle application-specific objects. Take the serialization of “getItemDetails” method as an example, there is an object *getItemDetailsSerializer* as the abstraction above. And it will break the method call into basic elements for further serialization. As shown in Figure 4.5, *GetItemDetailsSerializer* breaks the method into *authenticationData* and *info*, which is an exact mapping of the TPiSCAN specification.

```
public class GetItemDetailsMethodSerializer extends
    EfficientSoapSerializer<GetItemDetailsMethod>
implements IBodySerializer<GetItemDetailsMethod> {
    private void serializeMethod(GetItemDetailsMethod param) {
        this.getSerializer().startTag(NAMESPACE, "GetItemDetailsRequest");
        DataTypeSerializer.serializeIfcAuthenticationData(this, param.
            getAuthenticationData(), "authenticationData");
        DataTypeSerializer.serializeIfcItemDetailsInfo(this, param.
            getItemDetailsInfo(), "info");
        this.getSerializer().endTag(NAMESPACE, "GetItemDetailsRequest");
    }
}
```

Figure 4.5: GetItemDetailsSerializer.

On the one hand, the *GetItemDetailsSerializer* is an extension of *EfficientSoapSerializer* from "efficientsoap". From mobile TPiSCAN or AppIfcLibrary point of view, the *GetItemDetailsSerializer* hides the details inside efficientsoap for serialization, making the logic in code more clear. On the other hand, the hierarchy serializer matches nested data structure better. As in Figure 4.6, the uppermost object is an instance of a highly abstract data type, while the lowermost objects are instances of basic types.

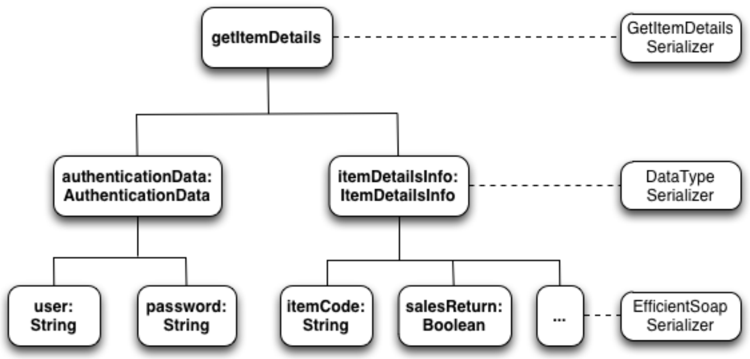


Figure 4.6: getItemDetails Hierarchy Structure.

4.2.2 Recursive Descent

Each layer of a complex data type has its own serializer. As in getItemDetails example, instances of AuthenticationData and ItemDetailsInfo will be serialized in DataTypeSerializer as in Figure 4.7.

```
public class DataTypeSerializer {
    public static void serializeIfcAuthenticationData(
        EfficientSoapSerializer serializer, IfcAuthenticationData
        ifcAuthenticationData, String tag) {
        serializer.getSerializer().startTag(null, tag);
        serializer.serializeStringElem(ifcAuthenticationData.getUser(), ``
            user'');
        serializer.serializeStringElem(ifcAuthenticationData.getPassword()
            , ``password'');
        serializer.getSerializer().endTag(null, tag);
    }
    // ... serializers for other data types
}
```

Figure 4.7: DataTypeSerializer.

And basic types like String and Integer will be serialized in EfficientSoapSerializer directly as in Figure 4.8.

```
public class EfficientSoapSerializer<T> {
    public void serializeStringElem(String str, String tag) {
        serializer.startTag(null, tag);
        serializer.text(str);
        serializer.endTag(null, tag);
    }
    //...
}
```

Figure 4.8: EfficientSoapSerializer.

This serialization strategy is consistent with the idea of recursive descent. The main goal is to serialize GetItemDetails object. But when the serializer encounters with elements of next layer (AuthenticationData and ItemDetails), it suspends current task and hands over the control to deeper serializers. After they finish the tasks, the serialization process will be called back to the initial object.

4.3 Thread Communication

When users use mobile phones, screen is the most important interface. To ensure thread-safe on Android platform, only main thread can access and update the user interface. When several threads in background need to interact with user interface, they have to be managed by the Looper object, queuing for the access in the message queue (see Figure 4.9). [11]

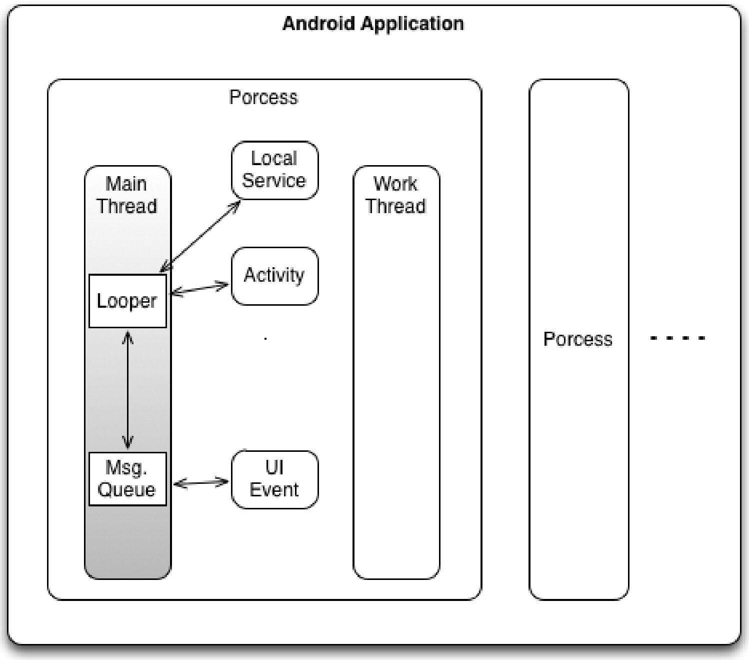


Figure 4.9: Insight of a Process.

It is clear that return values from work threads for deserialization or network communication cannot update user interface directly. Therefore, a "Listener Pattern" as in Figure 4.10 is used to notify the UI thread when the process is finished.

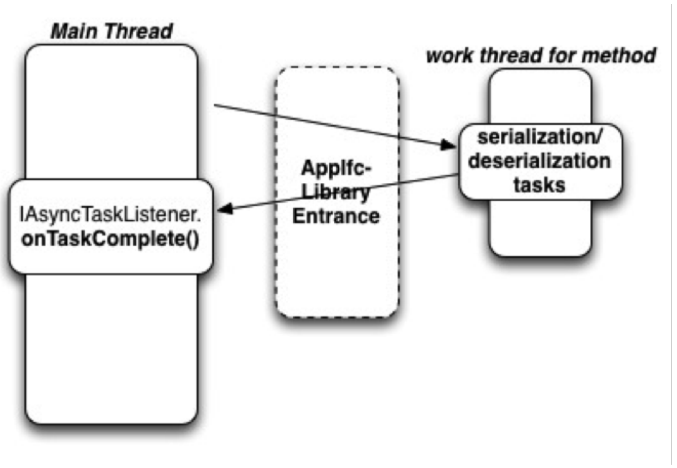

Main Thread
work thread for method
serialization/ deserialization tasks
Applfc-Library Entrance
IAsyncTaskListener. onTaskComplete()

Figure 4.10: Listener Pattern for Thread Communication.

IAsyncTaskListener in Figure 4.11 is one interface that notifies the main thread when the tasks in work thread are done.

```
public interface IAsyncTaskListener {
    public void onTaskComplete() throws InterruptedException,
        ExecutionException;
}
```

Figure 4.11: IAsyncTaskListener.

The application program has to implements this interface and provide the onTaskComplete() implementation for the library. So that the work thread can call back to the main thread (see Figure 4.12).

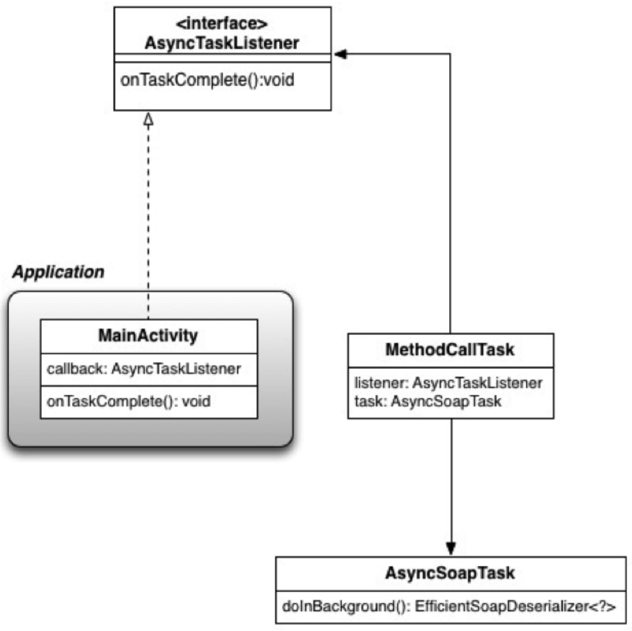


Figure 4.12: Listener Pattern Class Diagram.

Inside AppIfcLibrary, the getItemDetails from work thread will call the IAsyncTaskListener interface to update the graphic user interface (see Figure 4.13).

```
public void getItemDetails(IAsyncTaskListener listener,
        IfcAuthenticationData authenticationData, IfcItemDetailsInfo info) {
    GetItemDetailsMethod getItemDetailsMethod = new GetItemDetailsMethod()
        ;
    getItemDetailsMethod.setAuthenticationData(authenticationData);
    getItemDetailsMethod.setInfo(info);
    task = new GetItemDetailsTask(getItemDetailsMethod);
    task.execute();
    listener.onTaskComplete();
}
```

Figure 4.13: getItemDetails Callback.

The GetItemDetailsTask is an AsyncTask of Android operating system, dealing with serialization, Http connection and deserialization in the background. The execute() method will trigger the AsyncTask. And after the background computation finishes, the instruction pointer will jump back to the calling method.

4.4 In-bound Polling

The polling is used because of the method calls from server to clients (in-bound). The AppIfcLibrary uses 5-way communication in three times for polling, which logically realizes the bi-direction communication. The method "signatureRequired" from PosEventsGroup of TPiSCAN will be used as an example in the following.

4.4.1 Repetition

The polling needs to automatically repeat itself again and again. To do that, a Handler instance of the runnable thread is created and bound to current thread's message queue. As in Figure 4.14, the Handler object can help to schedule runnables to be executed.

```java
public class PollingThread implements Runnable {
    private String inBoundMethodName;
    private Boolean isExecute = true;
    PollPosMethod polling;
    private Handler mHandler = new Handler();
    private MethodReturnProvider methodReturnProvider = new
        MethodReturnProvider(this);

    public void run() {
        if (isExecute) {
            inBoundMethodName = polling.execute();
        }
        mHandler.postDelayed(this, 1000);
    }
    public void setIsExecute(Boolean isExecute) {
        this.isExecute = isExecute;
    }
}
```

Figure 4.14: Repetition of Runnable Thread.

The postDelayed() method set the runnable thread to repeate every 1 second; And the isExecute flag ensures that the thread will restart only after the finish of an incomming PosEventsGroup method if existed.

4.4.2 First Communication

At beginning, the AppIfcLibrary starts sending poll-quest to the server, asking whether there is a in-bound method from server to client (see Figure 4.15).

```
public class PollPosMethod {
    private PollingTask task;
    private String inBoundMethodName;
    private AppIfcLibrary adapter;
    public PollPosMethod(AppIfcLibrary adapter) {
        this.adapter = adapter;
    }
    public String execute() throws InterruptedException,
        ExecutionException {
        // ...
        task = new PollingTask();
        task.execute();
        if (!task.get().contentEquals("null")) {
            inBoundMethodName = task.get();
        }
        return inBoundMethodName;
    }

    public class PollingTask extends AsyncTask<URL, Integer, String>{
        //...backgroud task
    }
```

Figure 4.15: First Communication for In-bound Method Name.

The adapter provides the only interface for Android application and AppIfcLibrary to communicate with each other. The purpose of this communication is to get the name of in-bound method if existed. Only a “non-empty” response triggers following communications, otherwise the PollPosMethod just returns back and repeates runnable thread.

4.4.3 Second Communication

If there is an in-bound method call from server, the second communication will be triggered. This communication, as in Figure 4.16, picks up the input parameters from server. In “signatureRequired” example, that would be an instance of Message, explaining why the signature is required.

```
...
private AppIfcLibrary adapter;
private ISignatureRequiredReturnProvider listener = (
    ISignatureRequiredReturnProvider)methodReturnProvider;
if (!inBoundMethodName.contentEquals("nothing")) {
    setIsExecute(false); // stop polling
    if(inBoundMethodName.contentEquals("signatureRequired")) {
    InBoundMethodParamFetcher<SignatureRequiredMethod> fetcher = new
        InBoundMethodParamFetcher<SignatureRequiredMethod>(
        inBoundMethodName);
    SignatureRequiredMethod method = new SignatureRequiredMethod();
    fetcher.execute();
    method = (SignatureRequiredMethod)fetcher.get();
    adapter.getPosEventsGroup().signatureRequired(method.getMessage(),
        listener);
}
...
```

Figure 4.16: Second Communication for In-bound Parameters.

Knowing the name of a in-bound method, InBoundMethodParamFetcher in Figure 4.17 sends a serialized message to server. The return XML file will be reflected into an instance of BasicMethod after deserialization. Based on “thin-client” philosophy, the fist two serializations only produce simple string for SOAP message as a trigger for communication(see Figure 4.18).

```
public class InBoundMethodParamFetcher<R extends BasicMethod> {
    private String inBoundMethodName;
    private ParamFetcherTask task;
    private R response;
    public InBoundMethodParamFetcher(String inBoundMethodName) {
        this.inBoundMethodName = inBoundMethodName;
    }
    public R get() throws InterruptedException, ExecutionException {
        this.response = (R)task.get().getResponseObject();
        return response;
    }
    public class ParamFetcherTask extends AsyncTask<URL, Integer,
        EfficientSoapDeserializer<?>>  {
        public ParamFetcherTask() {}
        @Override
        protected EfficientSoapDeserializer<?> doInBackground(URL...
            params) {
        // Background tasks
    }
    ...
}
```

Figure 4.17: InBoundMethodParamFetcher.

4.4.4 Third Communication

The last communication sends the an serialized instance of Result as the output of inbound signatureRequired method. It serves as a confirmation to the server after a sucessful five-way communication. And the polling loop will be activated again by set the boolean flag in thread “true”.

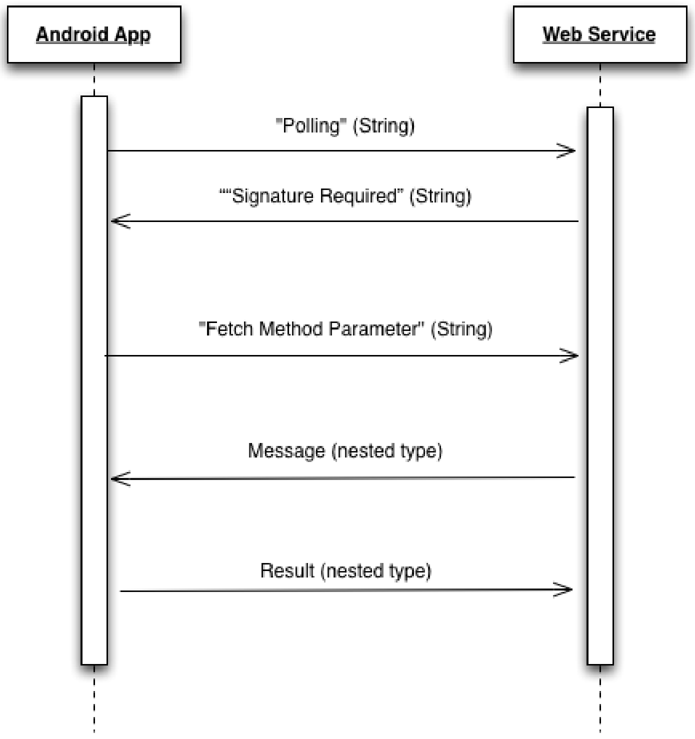


Figure 4.18: Sequence Chart for signatureRequired Method.

4.5 Library Structure

For clear boundary between Android application and AppIfcLibrary, there is a class:"AppIfcLibrary" as an adapter for the communication between App. and the library (see Figure 4.19).

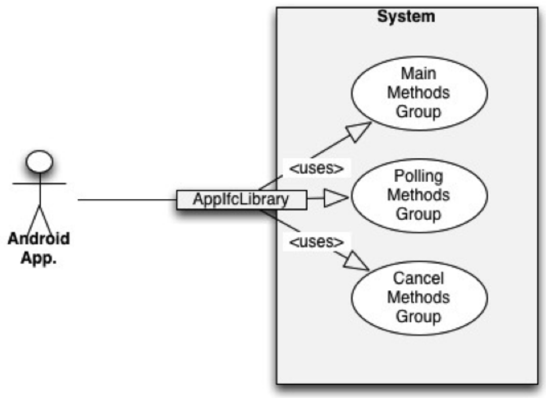


Figure 4.19: Diagram for AppIfcLibrary entry.

Different groups in the TPiSCAN interface have different agenda. "MainMethods-Group" is a group for out-bound methods from application (e.g., getItemDetails), while "PollingMethodGroup" is a group for in-bound methods from server (e.g., signatureRequired). The "CancelMethodGroup" is used to cancel method calls.

The advantage of a clear boundary is "AppIfcLibrary" abstracts the underlying techniques for serialization, deserialization and Http connection. The App.'s programmer only needs to call the methods through interface and focus on graphic interface and business logic.

In order to realize this adapter pattern, as in Figure 4.20, interfaces for different groups should register in the AppIfcLibrary.

```java
public class AppIfcLibrary {
    private IPosEventsGroup posEventsGroup;
    private IMainGroup mainGroup;
    private ICancelGroup cancelGroup;
    private PollingThread mThread = new PollingThread(this);
    public AppIfcLibrary(IPosEventsGroup posEventsGroup) {
        this.posEventsGroup = posEventsGroup;
        mainGroup = new MainMethodsCaller();
        cancelGroup = new CancelMethodCaller();
    }
    public ICancelGroup getCancelGroup() { return cancelGroup;}
    public IPosEventsGroup getPosEventsGroup() { return posEventsGroup;}
    public IMainGroup getMainGroup() { return mainGroup; }
    public void startCommunication(AppIfcLibrary lib) {
        Handler mHandler = new Handler();
        mHandler.postDelayed(mThread, 1000);
    }
    public void stopCommunication() {
        mThread.setIsExecute(false);
    }
}
```

Figure 4.20: AppIfcLibrary class.

And the Android application can call method getItemDetails from library by the command as follows:

library.**getMainGroup()**.getItemDetails(asyncTaskListener, authenticationData, info);

The asyncTaskListener is an instance of interface IAsyncTaskListener, which provides the access from worker thread back to UI thread."authenticationData" and "info" are the input arguments for the method.

The instances for IMainGroup and ICancelGroup are allocated inside AppIfcLibrary, while instance for IPosEventsGroup is allocated through a parameter. The reason is methods in IMainGroup are implemented by library, but methods in IPosEventsGroup should be provided by Android application because of the reversed direction.

The logic can be explained by "signatureRequired" (PosEventsGroup) example. It is the server that calls this method, asking signature from clients. Therefore, the App. should provide server an interface for the method and probably pop up a dialog to users. On the other hand, "getItemDetails" (MainGroup) is called by application. And the server is the callee in this case. In other word, the methods should be implemented and allocated memory in the "callee" of it (see Figure 4.21).

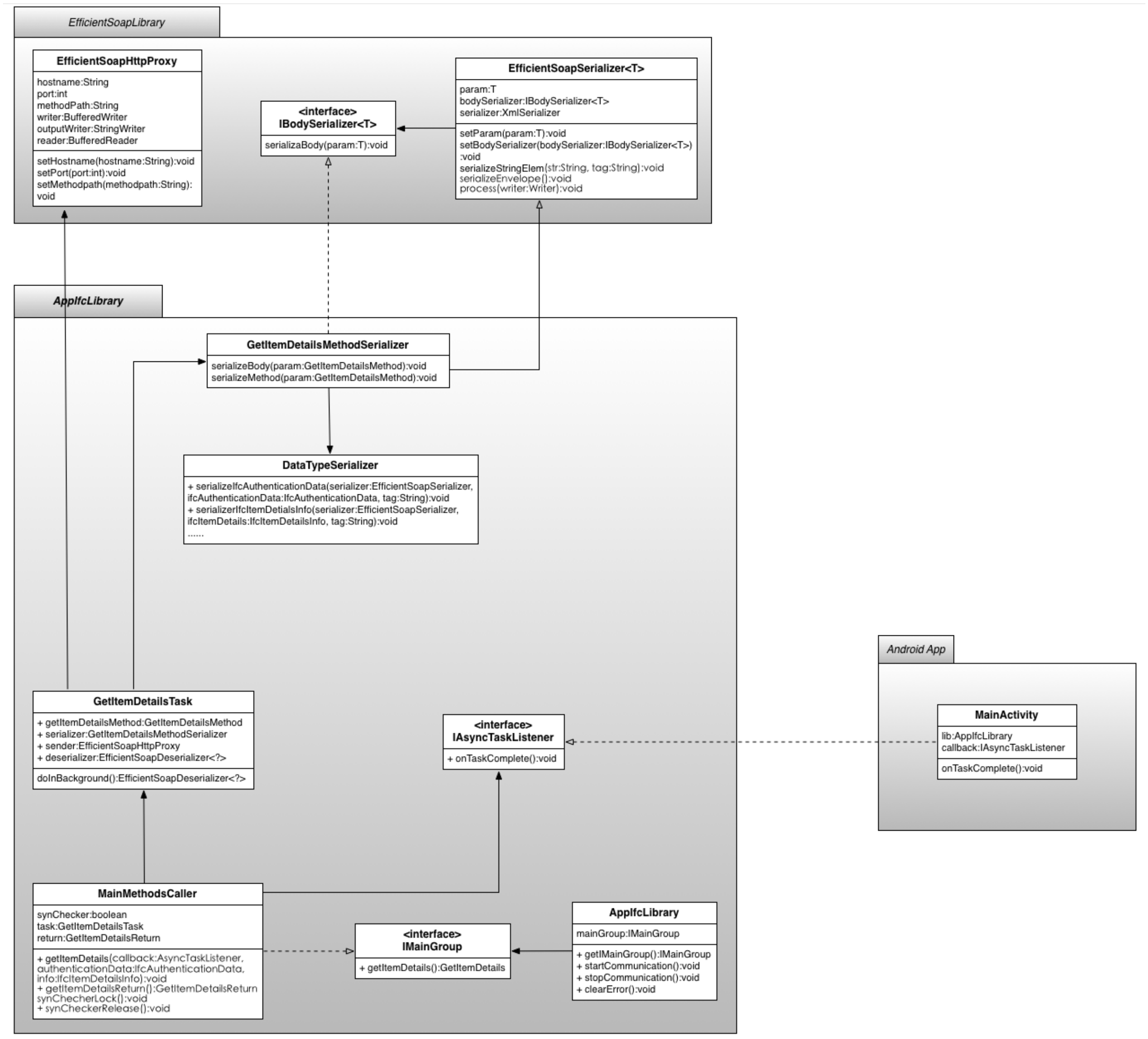

EfficientSoapLibrary
EfficientSoapHttpProxy
hostname:String
port:int
methodPath:String
writer:BufferedWriter
outputWriter:StringWriter
reader:BufferedReader
setHostname(hostname:String):void
setPort(port:int):void
setMethodpath(methodpath:String):
void
<interface>
IBodySerializer<T>
serializaBody(param:T):void
EfficientSoapSerializer<T>
param:T
bodySerializer:IBodySerializer<T>
serializer:XmlSerializer
setParam(param:T):void
setBodySerializer(bodySerializer:IBodySerializer<T>)
:void
serializeStringElem(str:String, tag:String):void
serializeEnvelope():void
process(writer:Writer):void
AppIfcLibrary
GetItemDetailsMethodSerializer
serializeBody(param:GetItemDetailsMethod):void
serializeMethod(param:GetItemDetailsMethod):void
DataTypeSerializer
+ serializeIfcAuthenticationData(serializer:EfficientSoapSerializer,
ifcAuthenticationData:IfcAuthenticationData, tag:String):void
+ serializerIfcItemDetialsInfo(serializer:EfficientSoapSerializer,
ifcItemDetails:IfcItemDetailsInfo, tag:String):void
......
GetItemDetailsTask
+ getItemDetailsMethod:GetItemDetailsMethod
+ serializer:GetItemDetailsMethodSerializer
+ sender:EfficientSoapHttpProxy
+ deserializer:EfficientSoapDeserializer<?>
doInBackground():EfficientSoapDeserializer<?>
<interface>
IAsyncTaskListener
+ onTaskComplete():void
MainMethodsCaller
synChecker:boolean
task:GetItemDetailsTask
return:GetItemDetailsReturn
+ getItemDetails(callback:AsyncTaskListener,
authenticationData:IfcAuthenticationData,
info:IfcItemDetailsInfo):void
+ getItemDetailsReturn():GetItemDetailsReturn
synChecherLock():void
+ synCheckerRelease():void
<interface>
IMainGroup
+ getItemDetails():GetItemDetails
AppIfcLibrary
mainGroup:IMainGroup
+ getIMainGroup():IMainGroup
+ startCommunication():void
+ stopCommunication():void
+ clearError():void
Android App
MainActivity
lib:AppIfcLibrary
callback:IAsyncTaskListener
onTaskComplete():void

Figure 4.21: Class Diagram for Serializer.

CHAPTER 5
RESULTS

This section will show a Android shopping application. The demonstration simulates a simple self-checkout shopping procedure. The user interface is shown in FIgure 5.1.

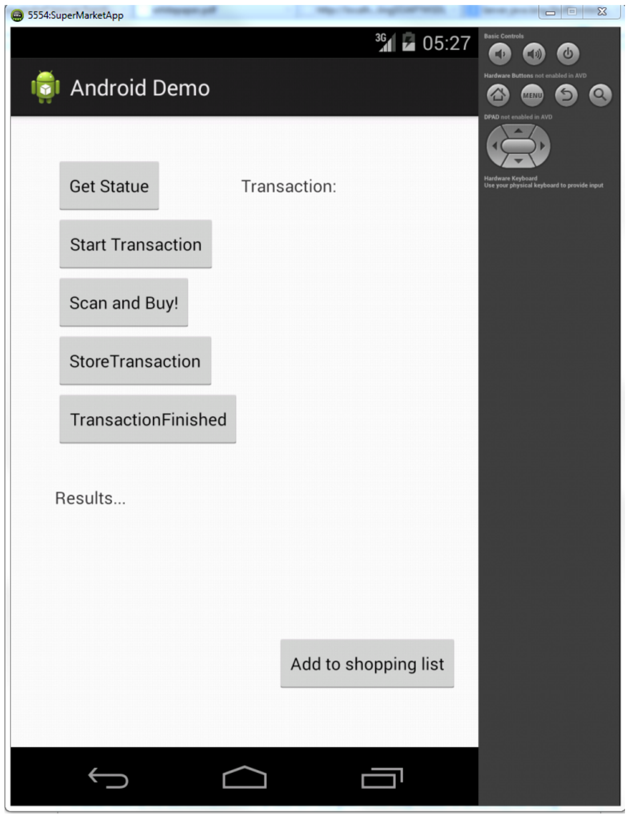


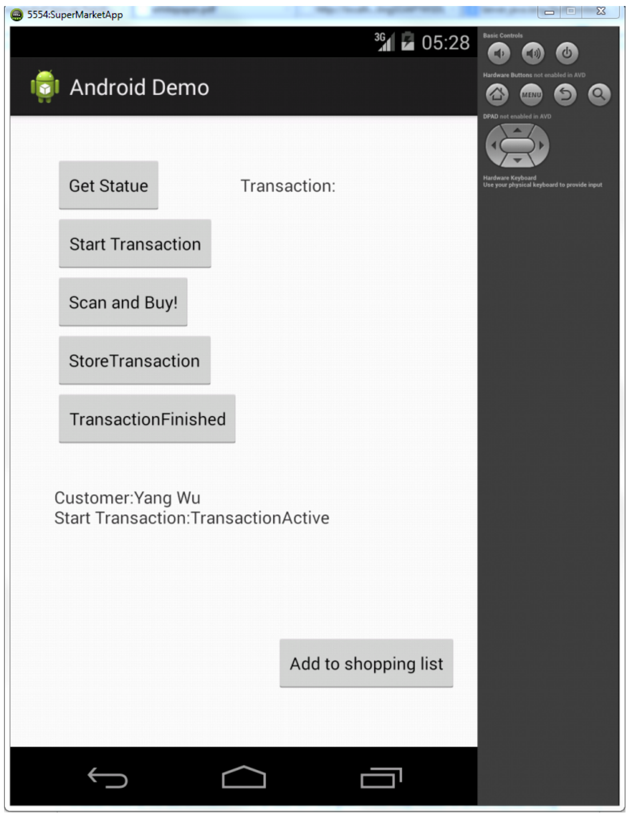
Figure 5.1: Android Application User Interface.

There are sei buttons (see Table 5.1).

Table 5.1: 6 Buttons for Android Demonstration.

Name	**Remark**
Get Status	fetch the current status of the POS application
Start Transaction	sets up a transaction at POS application for user
Scan and Buy	fetch detailed information about the item
Add to shopping list	add an item into the transaction
Store Transaction	ask the POS to store the current transaction for future activities
Transaction Finished	inform the POS that the self-checkout application has finished working on the transaction

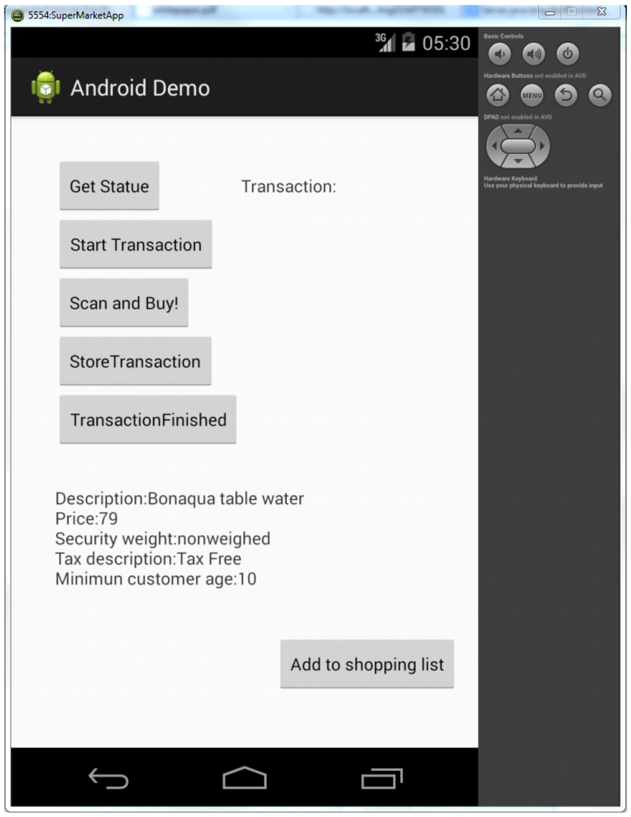
The 6 buttons illustrates a basic shopping procedure: a customer first checks the status of the current POS application and starts a new transaction, see Figure 5.2.



Customer:Yang Wu
Start Transaction:TransactionActive

Figure 5.2: A New Transaction.

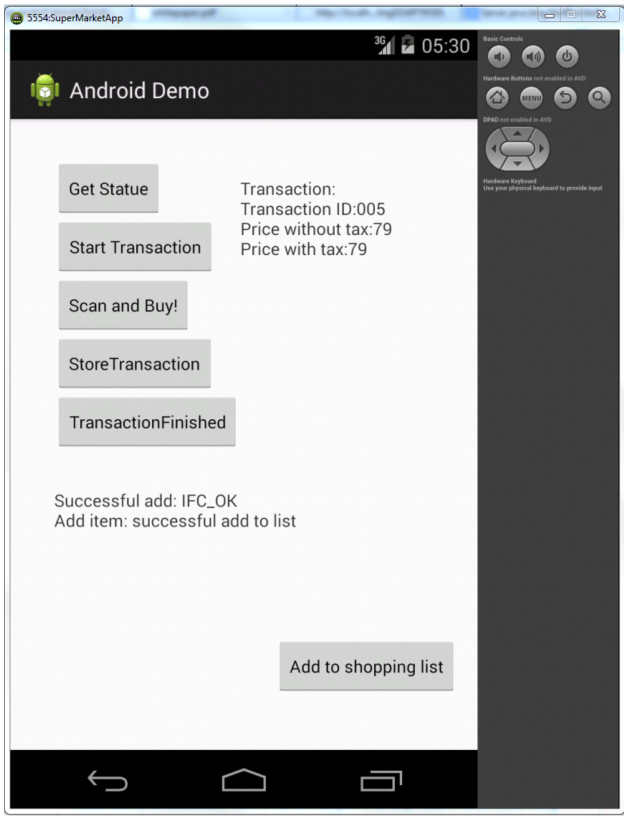
Then the customer scan items in the market as shown in Figure 5.3.



Description:Bonaqua table water
Price:79
Security weight:nonweighed
Tax description:Tax Free
Minimun customer age:10

Figure 5.3: Detailed Information about Scanned Item.

And if he or she were interested in the item, it can be added into the transaction. A virtual receipt (see Figure 5.4) will show up on the right side of screen. During shopping, the customer can also store the transaction in case of the break of connection. And in the end, the customer can finish the transaction.



Transaction:
Transaction ID:005
Price without tax:79
Price with tax:79

Successful add: IFC_OK
Add item: successful add to list

Figure 5.4: Add Item to Transaction.

For in-bound methods, the Android application should provide methods implemented them. In Figure 5.5, the App. provides method for "signatureRequired", poping up a dialog to customer.

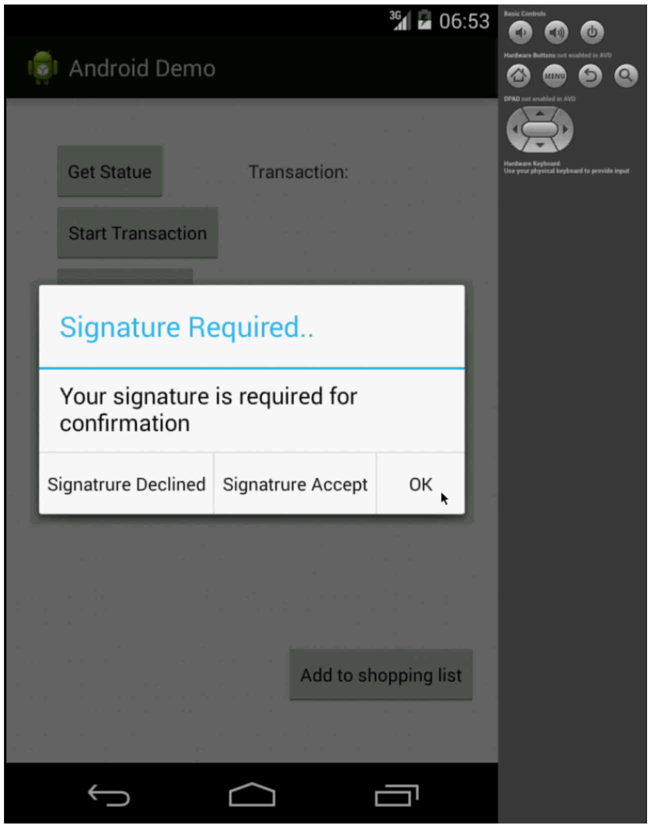
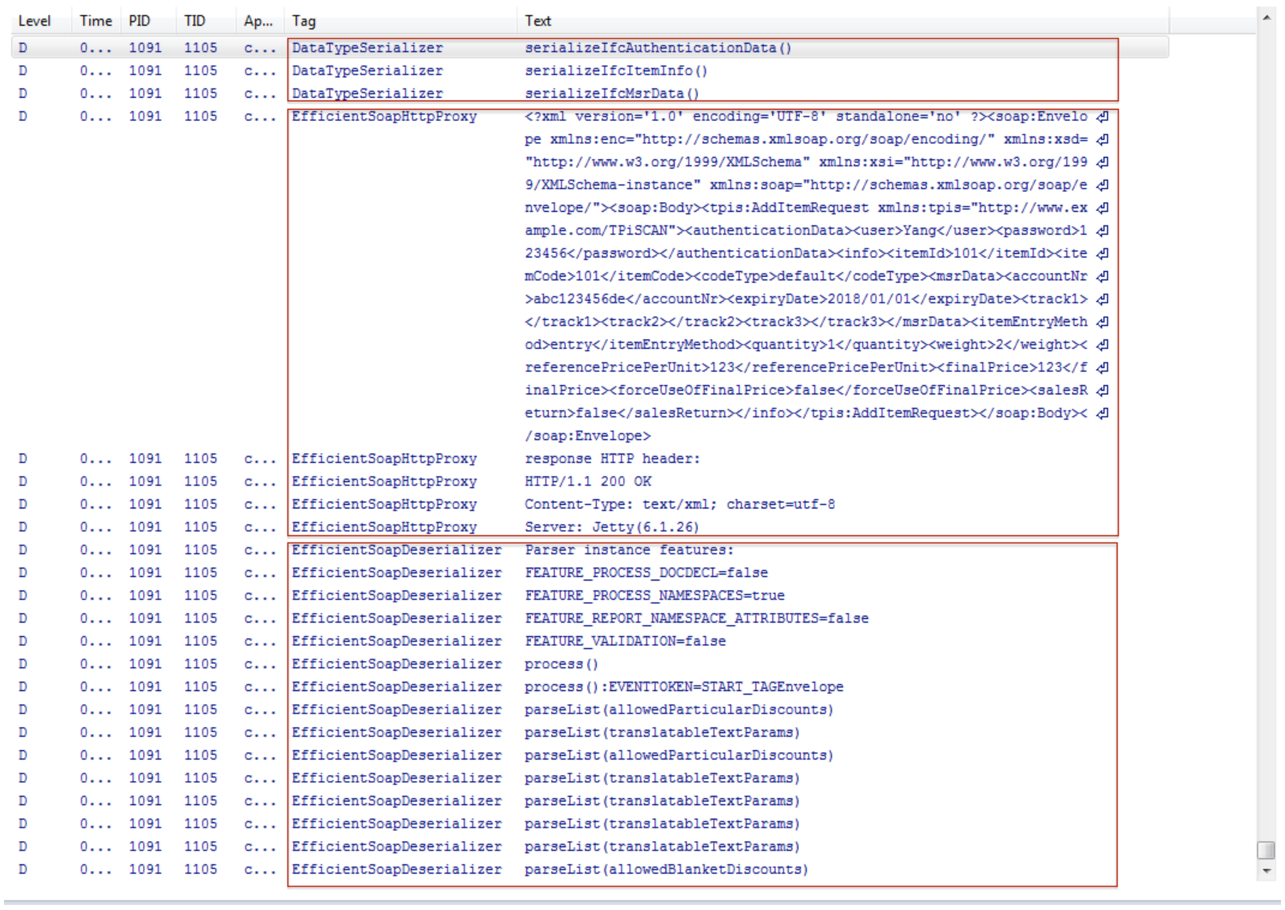


Figure 5.5: In-bound Method signatureRequired.

And behind these activities in foreground, it is “efficientsoap” that supports the application (see Figure 5.6). This is a screenshot for the log of method “addItem”. At the begining, the library calls methods from “DataTypeSerilizer” to serialize message. And then, a serialized XML object is sent through “EfficientSoapHttpProxy”. When the result returns from server, “EfficientSoapDeserializer” will help to parse the XML elements into objects.



```
Level  Time  PID   TID   Ap...  Tag                          Text
D      0...  1091  1105  c...   DataTypeSerializer           serializeIfcAuthenticationData()
D      0...  1091  1105  c...   DataTypeSerializer           serializeIfcItemInfo()
D      0...  1091  1105  c...   DataTypeSerializer           serializeIfcMsrData()
D      0...  1091  1105  c...   EfficientSoapHttpProxy       <?xml version='1.0' encoding='UTF-8' standalone='no' ?><soap:Envelo ⏎
                                                             pe xmlns:enc="http://schemas.xmlsoap.org/soap/encoding/" xmlns:xsd= ⏎
                                                             "http://www.w3.org/1999/XMLSchema" xmlns:xsi="http://www.w3.org/199 ⏎
                                                             9/XMLSchema-instance" xmlns:soap="http://schemas.xmlsoap.org/soap/e ⏎
                                                             nvelope/"><soap:Body><tpis:AddItemRequest xmlns:tpis="http://www.ex ⏎
                                                             ample.com/TPiSCAN"><authenticationData><user>Yang</user><password>1 ⏎
                                                             23456</password></authenticationData><info><itemId>101</itemId><ite ⏎
                                                             mCode>101</itemCode><codeType>default</codeType><msrData><accountNr ⏎
                                                             >abc123456de</accountNr><expiryDate>2018/01/01</expiryDate><track1> ⏎
                                                             </track1><track2></track2><track3></track3></msrData><itemEntryMeth ⏎
                                                             od>entry</itemEntryMethod><quantity>1</quantity><weight>2</weight>< ⏎
                                                             referencePricePerUnit>123</referencePricePerUnit><finalPrice>123</f ⏎
                                                             inalPrice><forceUseOfFinalPrice>false</forceUseOfFinalPrice><salesR ⏎
                                                             eturn>false</salesReturn></info></tpis:AddItemRequest></soap:Body>< ⏎
                                                             /soap:Envelope>
D      0...  1091  1105  c...   EfficientSoapHttpProxy       response HTTP header:
D      0...  1091  1105  c...   EfficientSoapHttpProxy       HTTP/1.1 200 OK
D      0...  1091  1105  c...   EfficientSoapHttpProxy       Content-Type: text/xml; charset=utf-8
D      0...  1091  1105  c...   EfficientSoapHttpProxy       Server: Jetty(6.1.26)
D      0...  1091  1105  c...   EfficientSoapDeserializer    Parser instance features:
D      0...  1091  1105  c...   EfficientSoapDeserializer    FEATURE_PROCESS_DOCDECL=false
D      0...  1091  1105  c...   EfficientSoapDeserializer    FEATURE_PROCESS_NAMESPACES=true
D      0...  1091  1105  c...   EfficientSoapDeserializer    FEATURE_REPORT_NAMESPACE_ATTRIBUTES=false
D      0...  1091  1105  c...   EfficientSoapDeserializer    FEATURE_VALIDATION=false
D      0...  1091  1105  c...   EfficientSoapDeserializer    process()
D      0...  1091  1105  c...   EfficientSoapDeserializer    process():EVENTTOKEN=START_TAGEnvelope
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(allowedParticularDiscounts)
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(translatableTextParams)
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(allowedParticularDiscounts)
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(translatableTextParams)
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(translatableTextParams)
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(translatableTextParams)
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(translatableTextParams)
D      0...  1091  1105  c...   EfficientSoapDeserializer    parseList(allowedBlanketDiscounts)
```

Figure 5.6: Log for Application.

REFERENCES

[1] C. Rau, *Effiziente Kommunikation über SOAP mit Android-Plattform-APIs*, Beuth Hochschule für Technik Berlin, 2013.

[2] Wikipedia, “Point of sale.” [Online]. Available: en.wikipedia.org/wiki/Point_of_sale

[3] S. Kannapinn, *TPiSCAN Interface Specification*, 2nd ed., Wincor Nixdorf, May 2012.

[4] J. Flanders, “Service Station: SOAP, REST and More,” *MSDN Magazine*, Jul. 2009.

[5] M. Novakovic, “Restlos glücklich?” *Java Magazin*, pp. 2–4, Mai. 2009.

[6] A. Silberschatz, P. Galvin, and G. Gagne, *Operating System Concepts.* Wiley, 2009.

[7] N. Wirth, *Theory and Techniques of Compiler Construction.* Addison-Wesley, 1996.

[8] S. Kannapinn, “Kistenwelt - strukturiertes und effizientes xml-parsing mit den apis sax2 und kxml,” *Javamagazin*, 2002.

[9] S.Kannapinn, “Systematischer Einsatz von Pull und Push-Parsern für XML,” in *W-JAX Conference*, Munich, Germany, Jan. 2002.

[10] S. Kannapinn, *Wincor Nixdorf's Code Generator for Self-Checkout Interfacing*, 1st ed., Wincor Nixdorf, Jun 2014.

[11] Wikipedia, “Thread.” [Online]. Available: en.wikipedia.org/wiki/Thread_(computing)