MASARYK
UNIVERSITY

FACULTY OF INFORMATICS

Local Network Communication in React Native

Master's Thesis

BC. DAVID LANG

Brno, Spring 2023

MASARYK UNIVERSITY

FACULTY OF INFORMATICS

Local Network Communication in React Native

Master's Thesis

BC. DAVID LANG

Advisor: prof. RNDr. Jiří Barnat, Ph.D.

Department of Computer Science

Brno, Spring 2023

FACULTAS ARTIS INFORMATICÆ • UNIVERSITAS MASARYKIANA •

Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Bc. David Lang

Advisor: prof. RNDr. Jiří Barnat, Ph.D.

Acknowledgements

I would like to thank my advisor prof. RNDr. Jiří Barnat, Ph.D., for valuable advice and guidance. I would also like to thank my family for their unwavering support throughout my studies.

Abstract

The thesis deals with communication over a local network between two React Native applications. It researches existing solutions and introduces a new open-source library to address the unmet requirements. The focus is on the development and design of the library. This thesis also covers the development of an application that serves as a customer-facing display in quick-service restaurants. This application demonstrates how the library can be utilized in the production environment.

Keywords

local network, open source, cross-platform development, Android, iOS, React Native, mobile application, customer-facing display

Contents

Introduction **1**

1 State of Art **2**
1.1 Requirements 2
1.2 Existing Solutions 3
1.3 Extending React Native Functionality 4

2 React Native Local Server **6**
2.1 TCP Layer 7
2.2 UDP Layer 11
2.3 Zeroconf Layer 12
2.4 Messaging Layer 14
2.5 Caller ID Layer 18

3 Figure CFD **21**
3.1 Figure Inc. 21
3.2 Customer Facing Displays 23
3.3 Goals and Requirements 24
3.4 Local Network Communication 25
3.5 Navigation 29
3.6 Monitoring 39

Conclusion **42**

Bibliography **43**

A List of Digital Attachments **44**

B List of Figure CFD Messages **45**

List of Figures

1.1 React Native Architecture [3] 5

2.1 RN Local Server - Package Diagram 6
2.2 RN Local Server - Sequence Diagram of TCP Classes . . . 9
2.3 RN Local Server - Object Diagram of TCP Classes 10
2.4 RN Local Server - Counter Screen 17
2.5 Whozz Calling? - Data Format [9] 18

3.1 Figure CFD - Use Case Diagram 25
3.2 Figure CFD - Example Message Sequence 28
3.3 Figure CFD - Navigation Tree 30
3.4 Figure CFD - Navigation Flow 31
3.5 Figure CFD - Setup Screen 32
3.6 Figure CFD - Welcome Screen 33
3.7 Figure CFD - Order Detail Screen 34
3.8 Figure CFD - Signature Screen 35
3.9 Figure CFD - Receipt Screen 36
3.10 Figure CFD - Thank You Screen 37
3.11 Figure CFD - Status Screen 38
3.12 Figure CFD - Bugsnag Stability Report 40
3.13 Figure CFD - Datadog Business Monitoring 41
3.14 Figure CFD - Datadog Communication Monitoring 41

Introduction

In today's digital age, software development has become an essential aspect of modern business. The rise of mobile applications has led to a significant increase in the use of cross-platform frameworks like React Native, which enables the development of high-quality, efficient mobile applications for both iOS and Android devices within a single codebase.

This thesis deals with the communication of two or more React Native applications over a local network. Even though it might seem there is no use case for such a communication, there are applications that can leverage it to provide better experience and stability to their users. The proof is that this thesis is motivated by a commercial company, Figure Inc., which wants to utilize local network communication in an application designed for restaurants. These industry-based applications are an excellent example, but besides that, this communication can be helpful for some mobile games or applications that deal with IoT devices.

The first chapter of this thesis researches existing solutions to local network communication in the framework React Native. Because there are no suitable libraries available, the second chapter presents a new library called React Native Local Server that aims to provide an interface allowing developers to implement local network communication in their applications with ease. The third chapter introduces the motivator of this thesis, Figure Inc., more in-depth. Furthermore, it covers the development of the application Figure CFD.

1 State of Art

This chapter aims to introduce the reader to the topic of local network communication in React Native. Firstly it presents the requirements for communication. Then it researches the existing solutions and validates them against the requirements. Furthermore, it explains how the functionality of React Native can be extended in a new library called React Native Local Server to meet the requirements.

1.1 Requirements

This section presents both functional and non-functional requirements for the library supporting local network communication.

- The library shall provide a TCP interface that allows opening a server socket on one device and connecting to it from other devices on a local network.
- The library shall provide a UDP interface that allows listening for UDP packets on a local network and sending those packets.
- The library shall provide a Zeroconf interface that allows registering a server as a service on a local network and browsing the available services.
- The library shall provide an abstraction above the networking protocols that allows programming communication between two or more applications declaratively.
- The library shall be compatible with both major mobile operating systems, Android and iOS.
- The library shall be written in TypeScript to provide type safety to its users.

1.2 Existing Solutions

Before creating a new React Native library, it is necessary to check if there is not already a library that would meet our requirements because React Native has a large community engaged in extending the framework's functionality. While researching libraries that deal with communication over a local network, we found the following options.

react-native-static-server[1]

This library allows serving static files from the device's file system. We might also need sharing of static assets between applications, but most importantly, we need dynamic communication in the form of requests and responses. Therefore this solution is not suitable for us.

react-native-http-bridge[2]

This library supports starting an HTTP server on the device, which might be an alternative to the TCP requirement. However, based on the time of the last published version, this library is no longer maintained.

react-native-sockets[3]

This library provides Java sockets on Android, allowing running a native socket server and client. It does not meet our requirements because it is not compatible with the operating system iOS and in addition, it is no longer maintained.

react-native-tcp-socket[4]

This library provides a socket interface for both Android and iOS with the support of SSL/TLS. It meets our TCP requirements, it is actively maintained, and based on weekly downloads, it is used by many projects. We considered using this library for the TCP implementation,

1. See `https://www.npmjs.com/package/react-native-static-server`
2. See `https://www.npmjs.com/package/react-native-http-bridge`
3. See `https://www.npmjs.com/package/react-native-sockets`
4. See `https://www.npmjs.com/package/react-native-tcp-socket`

but extending it with the Zeroconf protocols was impossible. Therefore, this library does not meet our requirements.

react-native-udp[5]

This library is similar to the one above but provides the UDP socket implementation. It is also actively maintained and has high weekly downloads, but extending it with the Zeroconf protocols is also impossible. Therefore, this library is not usable for us either.

Conclusion

Since none of the existing solutions would fit our needs, we have created a new library that extends React Native functionality with communication over a local network. We called the library React Native Local Server. It is published on NPM as @figuredev/react-native-local-server and available on GitHub at `https://github.com/FigurePOS/react-native-local-server`.

1.3 Extending React Native Functionality

When extending the functionality of React Native, there are two possible ways. The first option is to write the new functionality in JavaScript. It is relatively straightforward, but there is one significant limitation. This JavaScript code can use only APIs that React Native already exposes. If the React Native APIs are insufficient, there is the other option, which is called Native Modules. Native Modules in React Native are helpful if an application needs to access native platform API, which is unavailable in JavaScript by default [1].

A Native Module is an exposure of Java/Objective-C/C++ (native) classes to JavaScript as JavaScript objects, allowing execution of arbitrary native code from within JavaScript [1]. The implementation of the Native Module consists of two main parts: the JavaScript code and the native code. These two codebases communicate with each other through the so-called Bridge [2]. The interface of Native Modules provides three ways to invoke some native methods from JavaScript.

5. See `https://www.npmjs.com/package/react-native-udp`

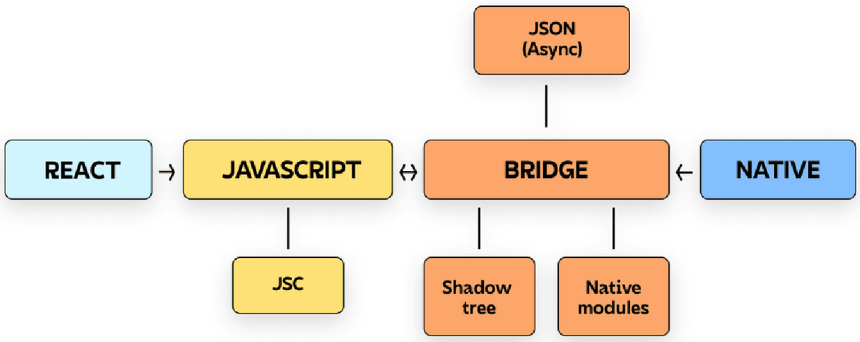

Figure 1.1: React Native Architecture [3]

The first option is a synchronous call. It is not a preferred option because it blocks the JavaScript thread, which makes the application unresponsive. The other option is to pass a callback function when invoking the method. The last alternative is to wrap the method's return value into a promise. The Native Module can extend an Event Emitter class, allowing it to send events that can be listened to from JavaScript. This is useful when the native code needs to communicate with JavaScript without being invoked.

The Native Modules are part of the legacy architecture of React Native, but they are still considered stable. An alternative to Native Modules in the new architecture is Turbo Native Modules. The main difference between those two technologies lies in the communication between JavaScript and native code. Native Modules communicate with JavaScript code through the so-called Bridge [2]. One of the most significant limitations of the Bridge is the need to serialize all data into JSON when transporting between JS and native code. In contrast, new Turbo Native Modules use a JSI (JavaScript interface), which removes the need to serialize all data into JSON [3].

At the time of React Native Local Server development, the new React Native architecture was still considered experimental and poorly documented. Hence the library uses the stable solution, Native Modules.

2 React Native Local Server

This chapter describes the design and implementation of the open-source library React Native Local Server. This library aims to expose networking protocols to JavaScript part of the application because the core of React Native provides only client-side APIs. The exposed API should be modular and on different levels of abstraction so that developers have more options to choose from to fit their needs.

For the initial release, five packages should be exposed from the library. The library can be extended in the future with more packages if needed. Three of those five packages port network protocols to JavaScript, requiring native implementation. The protocols are TCP, UDP, and Zeroconf. The other two aim to build an abstraction above low-level network protocols and are written in TypeScript.

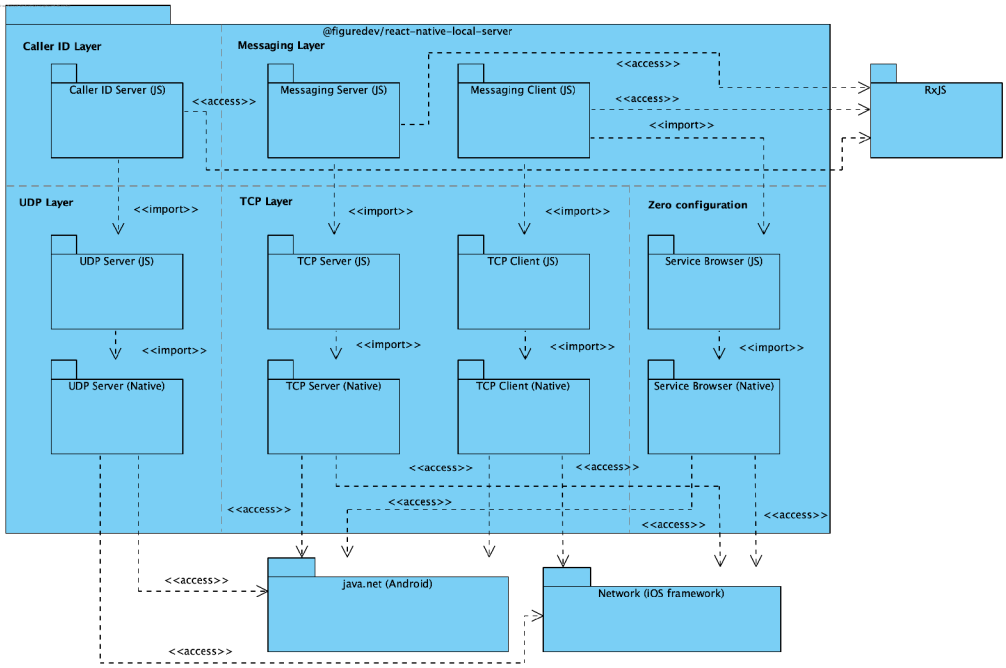

Figure 2.1: RN Local Server - Package Diagram

The library should have an example application that will demonstrate the usage of each exposed module. Another reason to have an example application is to speed up the development process because the application should be lightweight and thus fast to build. The last advantage of having the example application lies in debugging. If there is a bug in the library, it should be easily reproducible in the application. On the other hand, if the bug cannot be reproduced in the application, the library is most likely not the cause.

2.1 TCP Layer

The Transmission Control Protocol (TCP) is a widely-used protocol in computer networking that provides reliable, ordered, and error-checked data delivery between applications running on different hosts. It is a protocol that operates on the transportation layer in the TCP/IP model, and unlike UDP, it is connection-oriented. That means that there are two communicators, one is a server, and the other is a client. The communication has to be initiated by the client by contacting the server. However, at the same time, the server has to be ready to accept a new connection. Once the connection between the server and the client is established, they can send data to the other host. TCP guarantees that the data sent from one host will be received by the other host in the same order. This makes TCP an excellent choice for applications needing a reliable connection, such as World Wide Web, file transfer, and others. [4, Chapter 2.7.2]

The primary goal of the TCP layer in the library React Native Local Server is to allow a developer to establish a TCP connection between two applications without the need to write any native code. To achieve this goal, the public API of this layer consists of two TypeScript classes and related type definitions. The classes are TCPServer and TCPClient.

The class TCPServer provides functionality to start a TCP server on the device, manage connections and send data to those. Its constructor has a single parameter which is a string identifier. This identifier has to be unique, meaning no two servers can have the same id on the device.

After creating the instance, the server has to be started with the start method. The only argument of this method is a configuration object. It

consists of a port number and, optionally, service information used for Zeroconf protocol, see Section 2.3. Once the server is running, it can accept connections. Two types of events are emitted from native code when a new connection comes. The first indicates that the connection was accepted, and the other indicates it is ready to send and receive data. Both of these events have server id and connection id in their payload. The server id corresponds with the identifier provided to the server's constructor, and the connection id is randomly generated in native code using the UUID library.

Data can be sent to an active connection by the method sendData, which takes two arguments. The first argument specifies to which connection the data should be sent since there might be multiple open connections, and it is the identifier provided in events announcing the connection. The second argument is the data. It is possible to send only string data to simplify the first iteration. When new data arrives from some connection, the native code emits an event, which payload consists of the server and connection ids and the data.

If the connection is interrupted, native code emits an event with a payload of server and connection id and a reason. To close the connection programmatically TCPServer provides a method that takes two arguments, where the first one is the id of the connection and the second is a reason. This reason is later propagated in the event. The TCPServer also exposes a method that stops the whole server, meaning it cannot accept any new connections and closes any active ones. This method takes one argument, which is the reason to stop the server.

In addition, TCPServer also has multiple utility methods. Some methods return the ids of all open connections and the local IP address of the device. It is also possible to assign a logger to the server, which can provide valuable insights when debugging.

The purpose of the TCPClient is to connect to some TCPServer running on a different device on the local network. Its constructor also takes one argument, a unique identifier of the client on the device.

To start a connection, the server on the other device has to be ready, and the client needs to have the server's address. There are two methods of addressing the server. The first one is providing the server's IP address and port number. The other relies on the zero-configuration protocol where a service group and name specify the

address, see Section 2.3. Once the connection is established and ready to send and receive data, an event with the client id is emitted.

Sending the data to the server works the same as sending data from the server to the client. With a tiny difference, there is no need to provide a connection id because the client represents precisely one connection. When a client receives data from the server, the native code emits an event with a payload of the client id and the data itself.

Lastly, there is a method that stops the client and therefore closes the connection. If the connection is interrupted for any reason, an event with the client id and the reason is emitted to the JavaScript.

A simplified example lifecycle of TCPServer and TCPClient is shown in Figure 2.2.

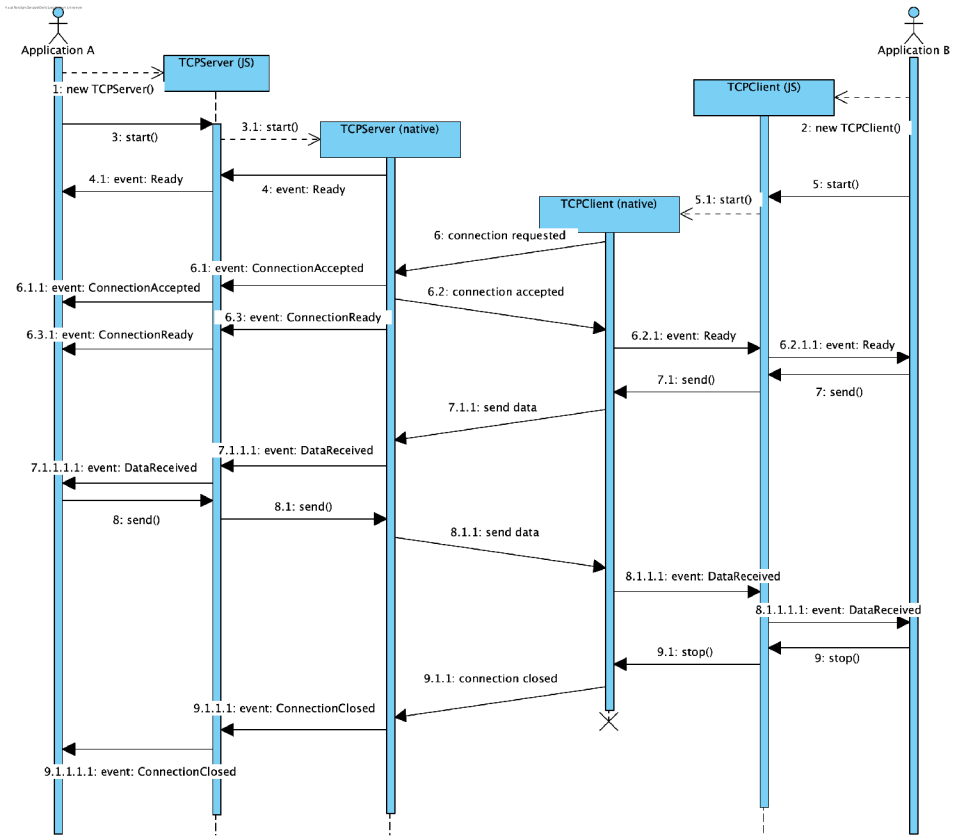

Figure 2.2: RN Local Server - Sequence Diagram of TCP Classes

The TCP layer consists of two native modules, one for the server and the other for the client. For the Android implementation, we decided to leverage the core java.net package. Where the server uses ServerSocket class and the client uses the Socket class. On iOS, we utilized the core Network framework. The server implementation lies on NWListener, and the client uses NWConnection under the hood.

One of the limitations of native modules in React Native is that there is always exactly one instance of the native module running at a time. Therefore it is impossible to spawn new instances of the module dynamically. If we would consider the native module a class implementing the TCP protocol, it limits us to having just one TCP server or client running at the same time. This constraint might be a blocker for some applications because it is plausible that some applications must run multiple servers simultaneously, each with a different purpose. To solve this problem, we decided to consider the native module class a manager. Its responsibility is to manage multiple instances of server classes based on requests from the JavaScript part of the application. This design applies the same way to the client implementation. For more information, see Figure 2.3.

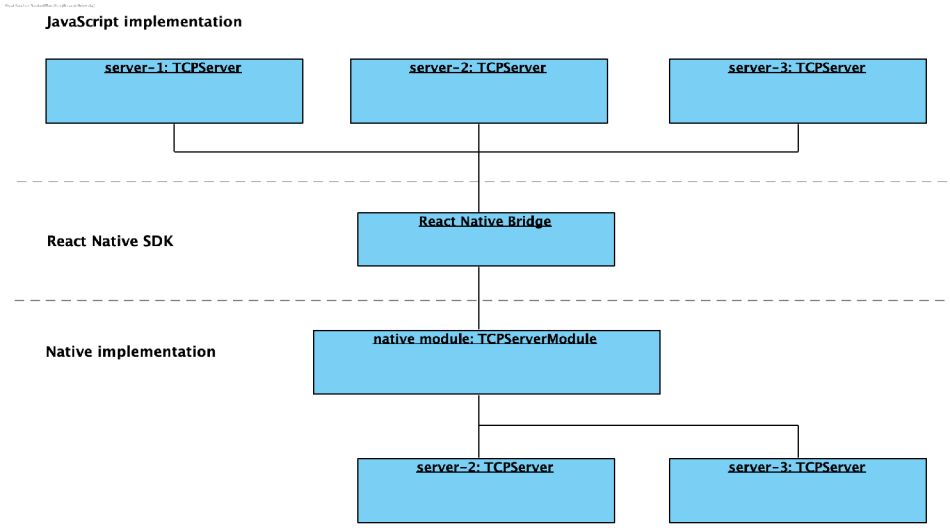

Figure 2.3: RN Local Server - Object Diagram of TCP Classes

The example application packaged with the library has two screens demonstrating the usage of the TCP layer. The first screen allows the user to start a local TCP server on provided port number. It also shows the current state of the server and all active connections. The user can stop the whole server or close individual connections from this screen. The active connections are displayed as tabs. After pressing the connection, the user can send any string data to the client and see all incoming and outgoing data related to this connection. The second screen allows the user to connect to a device in a local network with a running TCP server by providing its IP address and port number. The user can also see the client's current state and stop it if needed. The last functionality of this screen lies in sending data to the server and seeing all incoming and outgoing data as well.

2.2 UDP Layer

The User Datagram Protocol (UDP) is a lightweight, connectionless protocol used for transmitting data over a network. Unlike TCP, which provides reliable, ordered, and error-checked delivery of data, UDP does not guarantee the delivery of data or the order in which it is received. However, this simplicity also makes UDP more efficient than TCP in some situations, such as for real-time applications like video streaming or online gaming, where speed is more important than reliability. [4, Chapter 2.1.4]

The library React Native Local server provides one simple class that aims to expose UDP server and client functionality to JavaScript code in React Native applications. This class is called UDPServer, and its constructor takes one argument, a unique identifier of the server on the device. After instantiating the class, the server can listen to incoming UDP packets. The start method takes a configuration object as the first argument. The configuration object consists of a port number. Any incoming data are announced by emitting an event with the server id and the data in string format. If the user wishes not to listen for incoming packets anymore, they can call a stop method with one optional argument, which is a reason why the server should stop.

Unlike TCPServer, the UDPServer can send data even when not running. This behavior is possible thanks to the fact that UDP is con-

nectionless. Therefore the class UDPServer can be used as a UDP client to send data. The send method requires three arguments. The first argument is the IP address of another UDP server on a local network, and the second is its port number. The last argument is the data that should be sent. The UDPServer supports sending only string data for now, as well as TCPServer and TCPClient.

The library creates one native module to provide the UDP functionality. This module follows the principles used for the other two native modules, meaning it is a manager that dynamically spawns new instances of the native UDP server class. The iOS version uses the same class (NWListener) as the TCPServer but with different properties. The Android version again leverages the package jave.net, and this time it uses DatagramSocket class. The native module has to accommodate the need for sending data without having any server class running. This requirement is fulfilled by spawning a one-time socket, closed right after the data are sent.

One screen is dedicated to the demonstration of the UDPServer in the example application. On this screen, users can start a UDP server on a port of their choice, afterward stop the server, and see its current state. Once the server is running, all incoming data are displayed in the middle section of the screen. Users can also send data to another server after specifying its IP address and port number.

2.3 Zeroconf Layer

The Zero Configuration Protocol (Zeroconf) is a set of networking protocols designed to automatically configure IP-based networks without requiring manual configuration or additional servers. The primary goal of Zeroconf is to allow devices to discover and communicate with each other without any need for user intervention or network administration. In other words, Zeroconf allows services to advertise themselves on a local network. Clients can then browse desired services and connect to them without knowing the service's IP address and port number beforehand. This makes any product that uses Zeroconf user-friendly, even for users not experienced with network configuration.[5]

The purpose of the Zeroconf layer in the library React Native Local Server is to provide an interface thanks to which clients can discover related services implemented with this library. It should allow developers to create user-friendly applications for their users. That interface is implemented by a class called ServiceBrowser. In Zeroconf, each service is addressed by its human-friendly name and service group [5, Chapter 4]. The ServiceBrowser class allows users to browse for precisely one service group. There might be applications that will need more than one browser. Therefore every ServiceBrowser has to have its unique identifier, which is provided as the only argument of its constructor.

Since the ServiceBrowser listens for changes in the provided service group, it has only two simple methods. The first method starts the browser, and the second method stops it. The configuration object passed to the start method specifies the desired service group. Whenever the browser discovers a new service, it emits an event with its id and information about the service, meaning its name. When some service becomes unavailable, the browser announces it byt an event with the same data in its payload.

The Zeroconf layer is not that useful by itself, but it becomes powerful when used with the TCP layer. When starting a TCPServer, the configuration allows specifying a service group and name, which makes it later discoverable by the ServiceBrowser. If the Zeroconf configuration is provided to the server, it automatically advertises itself once ready and unregisters itself when stopped. In addition, the TCPClient can be started with the service group and name discovered by the ServiceBrowser instead of an IP address and port number. This feature brings two advantages to the developers and their end users. Firstly, the end user does not have to fill in the IP address, which makes the application nicer to use. Moreover, the servers can run on dynamic port numbers, which mitigates the risk that some other application on the device already holds the desired port.

The Android implementation of the ServiceBrowser again leverages the package java.net, precisely a class called NsdManager. The iOS is implemented by using NWBrowser from the Network framework.

The example application demonstrates the usage of ServiceBrowser on a single screen. The user can input a desired service group and

then start the browser. All available services are then displayed on the screen.

2.4 Messaging Layer

The messaging layer of React Native Local Service should provide a declarative interface that allows developers to implement communication over a local network between two React Native applications with ease. The layer is called messaging because cloud messaging systems inspired it, and the smallest data unit is a message.

The communication between the applications should be reliable, meaning all messages will be received in the same order they were sent, and no message will be lost. Therefore this layer is built on top of the TCP layer and follows the client-server model. As in the TCP layer, two main classes are exported from this layer: MessagingServer and MessagingClient. Both classes integrate with the Zeroconf layer to allow developers to create user-friendly applications. When starting the server, the user can provide a service configuration, which causes the server to register as a service. The client can then listen for some service group and connect to the desired server without knowing its IP address and port number.

To achieve the declarative interface when dealing with asynchronous code, we used a JavaScript library for reactive programming called RxJS. It provides tools and utilities for working with asynchronous data streams. It is built on top of the Observable and Observer patterns and allows developers to manage complex data flows concisely and declaratively. RxJS provides many operators and functions for working with observables, such as map, filter, merge, throttle, and debounce. These operators allow developers to transform, filter, combine streams, and apply various data processing and manipulation techniques. One of the key benefits of RxJS is its ability to create custom operators and use them in any stream [6].

The core concept of the messaging layer is called a message handler. This concept is used in both MessagingServer and MessagingClient. Message handlers allow developers to write the communication logic declaratively. This concept was inspired by a library called Redux Observable and its Epics [7]. Technically a message handler is a pure

JavaScript function that takes two arguments. The first argument is an RxJS stream of all incoming messages, and the second argument is an object including dependencies. These dependencies are defined by the developer when starting the server or client. It can contain functions that allow accessing or modifying the current state of the application. For now, the handler should always return an empty stream because it is still being decided on the best use of the output stream. Message handlers are a powerful tool because they can handle different messages the same way or act only on some sequence of messages. Listing 2.1 shows one of the handlers from the example application. Both MessagingServer and MessagingClient require exactly one message handler in their start method. To accommodate this requirement, the messaging layer exposes a function that combines multiple handlers into one, usually called the root handler.

Listing 2.1: RN Local Server - Example Handler

```
export const counterRequestedHandler = (
    message$,
    deps
) =>
    message$.pipe(
        ofMessageType(CounterMessageType.CountRequested),
        switchMap((request) => {
            const count = getCounterCount(deps.getState())
            CounterServer.send(
                createCounterMessageCountChanged(count),
                request.source.connectionId,
            )
            return []
        })
    )
```

As mentioned before, the smallest data unit of messaging layer is a message. Each message has its identifier, timestamp, information about the source, and body. The body can be any JavaScript object that can be serialized to a JSON string and is defined by the developer. Besides messages, the server and client can exchange ping requests and responses to ensure the other application is still running and responsive. The server sends ping requests to the client in a config-

urable interval, and for every request, it waits a specified time for any incoming data. If the request timeouts, meaning there are no new data in the specified time, multiple times, the server closes the connection with timeout reason. Every time the client receives the ping request, it replies with a simple ping response. The client closes the connection with a timeout reason when there are no new data, meaning messages or ping requests, in the configurable time. The ping functionality is beneficial when there is a risk that the devices will leave the network because the native implementation of TCP does not detect this scenario.

The MessagingServer class is similar to the TCPServer. Its constructor takes exactly one argument: the unique identifier of the server in the application. The start method takes a configuration object as its first argument. This object consists of the port number, ping parameters, and service information. The second argument of the start method is the root handler, and the last argument is the dependency object that is passed to every handler. Similar to the TCP server, the messaging server can be stopped by a method that takes one optional argument, which is the reason for stopping the server. Unlike the TCP server, the messaging server provides a getter for a stream of all status events, such as new connections and interruptions. Also, the send method does not accept a string as the data but rather the message body, which can be any serializable object. Another difference between the messaging and the TCP server is that all methods of messaging server return RxJS observables rather than promises.

The MessagingClient class follows the principles of MessagingServer. The start method has three arguments: the configuration object, the root handler, and the dependency object. It also has its unique identifier, but its constructor takes one additional argument: the name of the service group. Then there are two methods that allow to start and stop the service search. The notifications about new and lost services are pushed into an stream. When the client and server use the Zeroconf functionality, the client flow will usually begin by starting the service search, and once the desired service is found, the client will connect to it.

There are three screens dedicated to the messaging layer in the example application. Two of them copy the TCP screens. The user interface looks the same, but it is powered by the messaging classes.

The reason for this is to demonstrate the difference between those two layers and their advantages. The third screen aims to illustrate the use of messaging layer for synchronization purposes. The screen is called a counter, and unlike other screens, this screen is shared between the server and the client. The main section shows a simple counter with increase, decrease, and reset buttons. In addition, there is a button that starts and stops automatic incrementation. The primary goal of this screen is to display a synchronized counter on multiple devices. One of the devices is a server, the source of truth, and the others can connect to it as clients. They can connect to the server by providing the server's IP address and port number or using service discovery. While the screen is in client mode, the counter is synchronized with the server, and the only action the user can take is to request the server to reset the counter. Figure 2.4 displays the counter screen in the client mode.

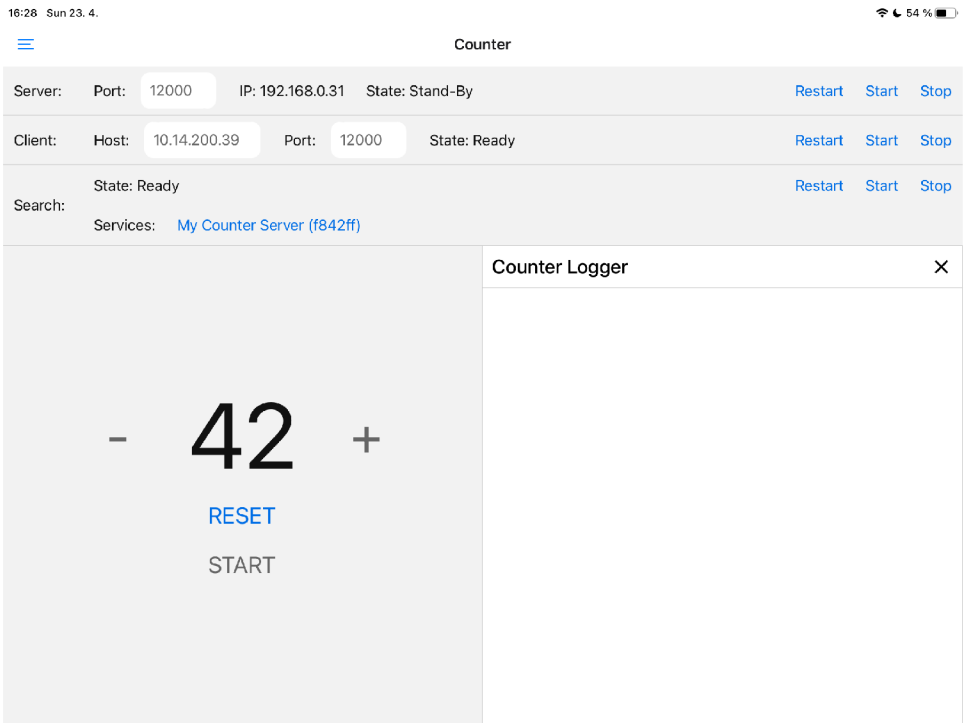

Figure 2.4: RN Local Server - Counter Screen

2.5 Caller ID Layer

Caller ID is a service feature that identifies incoming callers before the call is answered. It functions by displaying the phone number or name of the caller on the recipient's phone screen, assuming that the information is available and not blocked by the caller. Caller ID has become a ubiquitous feature on modern telephony systems, serving as a valuable tool for screening incoming calls and avoiding unwanted solicitations or potential scams. By providing the name or phone number of the caller, recipients can make informed decisions about whether to answer the call, let it go to voicemail, or block it altogether [8].

One of the caller ID applications is a device called Whozz Calling. This device provides caller information from line and vertex phones. It connects to the phone and IP network, and on every incoming call, it broadcasts UPD packets to the network with the caller information. The packets are fixed field, space delimited, and ASCII text[9]. See Figure 2.5 for more information about the format. Then a computer listens for those packets and displays the information on its screen. The information is usually used to prefill some form in the application operated by the end-user when answering the phone.

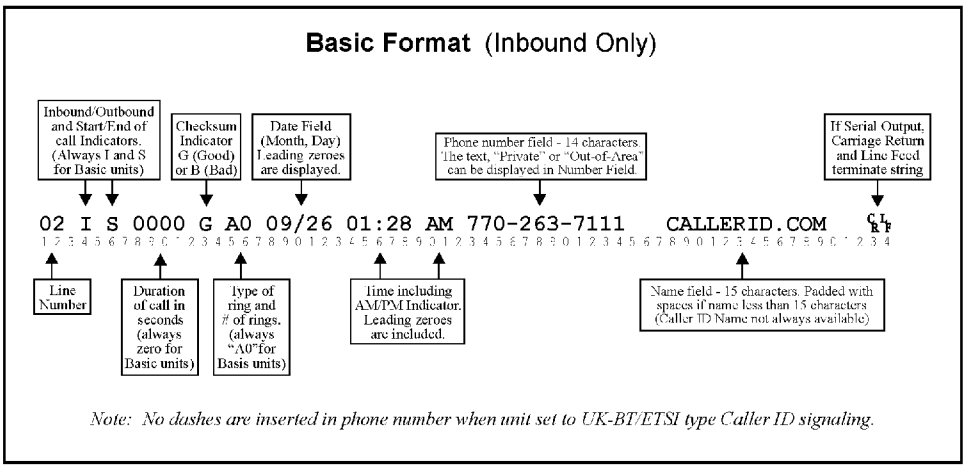

Figure 2.5: Whozz Calling? - Data Format [9]

The Caller ID layer in React Native Local Server provides a declarative interface that simplifies call detection and allows developers to focus on their application rather than the technical details of Whozz Calling devices. It provides a single class called CallerIdServer. The class has methods that start and stop the server and a getter that returns an RxJS stream of all server lifecycle events. All information from the Whozz Calling device is parsed and pushed to another stream. The information is shared as a plain JavaScript object. See Listing 2.2 for its TypeScript definition.

Listing 2.2: CallerIdServer Type Definition of Call Object

```
export type PhoneCall = {
    number?: "Private" | "Out-of-Area" | string
    name?: string
    line?: string
    date?: string
    time?: string
    direction?: "inbound" | "outbound"
    update?: "start" | "end"
    duration?: number
    checksum?: "good" | "bad"
}
```

The CallerIdServer is built on top of the UDPServer class and utilizes the RxJS library. The CallerIdServer creates an RxJS from all data incoming to the UDP server and processes them in the stream pipeline to provide the data in the desired shape. Because the UDP protocol is not reliable, the Whozz Calling device sends multiple packets for a single call to increase the probability of the packets being successfully delivered. Therefore, the packets arriving at the CallerIdServer must be deduplicated by their content and time. After the deduplication, the server checks the data format and parses them. Lastly, the data are filtered based on the checksum provided by the Whozz Calling device and the phone call direction. See the simplified pipeline in Listing 2.3.

Listing 2.3: CallerIdServer Data Pipeline

```
fromUDPServerEvent(this.serverId, "DataReceived").pipe(
    log(`CallerIdServer [${serverId}] data received`),
    map((event) => event.data),
    deduplicateBy(identity, deduplicationTime),
    log(`CallerIdServer [${serverId}] data deduplicated`),
    map((data) => [
        parsePhoneCallFromPacketData(data),
        data,
    ]),
    filter(([call, data]) => {
        if (call == null) {
            this.logger?.error(`Invalid Data`, data)
            return false
        }
        return true
    }),
    map(([call, _]) => call),
    log(`CallerIdServer [${serverId}] - call parsed`),
    filter(hasPhoneCallGoodChecksum),
    filter(isPhoneCallInbound),
    log(`CallerIdServer [${serverId}] - call detected`),
    share()
)
```

The developer can see an example usage of the CallerIdServer in the example application. There is one screen dedicated to it. On this screen, the user can start and stop the server and see its current state. All incoming calls and their history are displayed in the middle main section of the screen. The screen also allows the user to simulate the call to ease the testing without the actual device.

3 Figure CFD

This chapter centers around the US company Figure Inc., which inspired the thesis. A customer-facing application called Figure CFD needs to be developed as part of the assignment. The purpose and meaning of a customer-facing application are explained, and goals and requirements for Figure CFD are established. The chapter then delves into the utilization of React Native Local Server library for the application, followed by a detailed description of each screen of the final application. The chapter concludes by discussing how the application monitoring was addressed and presenting the corresponding measured values.

3.1 Figure Inc.

Figure Inc. is a US company that aims to help small and medium businesses with their growth. Figure focuses mainly on restaurants, bistros, fast foods, and other gastro businesses, but in the future, they aim to enter a retail market and more. Figure's mission is to save business owners valuable time so they can spend it on things that matter. Business owners can save time by automating many steps, which they have to do regularly. Figure also helps to grow the business by providing valuable insights. Owners and managers can get various reports to help them decide their business's next steps.

The company Figure Inc. is a successor of Two Beards, LLC., founded in 2015 in New York by two men who studied together, Jin Woo Park and Ladislav Červinka. The first product the company featured is an iPad-based application called Figure POS. After that, there was another product, this time it was a web-based application designed to simplify configuration. The company Figure Inc. has grown the most in the years 2020 and 2021. The reason for that was the pandemic of COVID-19 and Figure's ability to help restaurants sell their food on the online ordering app, Make It Butter. Many merchants who started using Make It Butter during the pandemic also decided to use Figure POS afterward.

Figure Inc. features several products to fulfill its mission. Most of these applications are for iOS, but there are also two web applications as well as one Android application.

The application Figure POS[1] is the core of Figure's ecosystem for small and medium businesses. It is an iPad based point-of-sale application. It was initially released on App Store in 2017 and is still actively maintained and developed. Merchants use this application to create and track orders in their business. They can also process card payments through it, accept tips, and make refunds if needed. Figure POS integrates with Star printers, allowing users to print sale receipts for end customers and kitchen receipts that instruct kitchen staff what meals they should prepare. Figure POS targets mainly restaurants and other gastro businesses. These businesses can leverage an extensive table management view. This view provides valuable information to the servers, such as the number of guests at each table, seating time, and more. Overall, Figure POS is a modern point-of-sale system that is very easy to use and has excellent visuals.

Besides Figure POS, merchants can use an online ordering application, Make It Butter[2], which helps them with their online presence and selling their products online. The last application that allows merchants to accept more orders is Savor Kiosk[3]. It is an iPad-based application placed in the restaurant as well as Figure POS. However, unlike Figure POS, it is operated by the end customers. Then there are two applications designed for restaurant managers, Figure Manage[4] and Figure Mobile[5]. The last application in the ecosystem is Figure KDS[6]. It is an application developed for the kitchen, and the kitchen staff can see which meals they should prepare.

1. See `https://apps.apple.com/us/app/figure-pos/id1040969507`
2. See `https://www.makeitbutter.com/`
3. See `https://apps.apple.com/us/app/savor-kiosk/id1453050116`
4. See `https://app.figurepos.com/login`
5. See `https://apps.apple.com/us/app/figure-mobile/id1552215135`
6. See `https://apps.apple.com/us/app/figure-kds/id1518241630`

3.2 Customer Facing Displays

The customer-facing displays are screens pointed toward the customer in a store. These screens are designed to be less interactive than self-ordering kiosks. While kiosks are here to replace the servers behind counters, customer-facing displays aim to improve the checkout process for both merchants and customers. The customer-facing displays can be seen mainly in quick service restaurants, which need to sell as many orders in the shortest time possible.

There are four ways how a customer-facing display can help a restaurant [10]. The main benefit of using a customer-facing display is reducing misunderstandings during checkout. While ordering, the customers can see all the products they have ordered, including various modifications. Customers can also see the price breakdown for the whole order to know which taxes, fees, and discounts are applied. Providing all this information in time mitigates chargebacks.

The second reason to use a customer-facing display in a restaurant is to speed up the whole checkout process. If the customer-facing display is touch-enabled, customers can interact with the screen. For example, they can choose the amount they want to tip. The customer-facing display can also be used to sign the payment. This is relevant mainly for countries where people are used to signing a receipt after payment, like the USA.

The merchant can use a customer-facing display to promote some of his products on the screen. These promotions aim to increase customers' purchase of products and ultimately increase the merchant's revenue.

Last but not least, customer-facing displays can enhance a loyalty program. The display can show, for example, how many points the customer got for that order. Furthermore, if the customer does not use the loyalty program, the display can motivate them to join it.

3.3 Goals and Requirements

This thesis aims to develop a customer-facing display called Figure CFD. The primary motivation of Figure Inc. to introduce the new application is that most of Figure's competitors already have a customer-facing display. This application will help Figure be a more attractive option for quick-service restaurants.

Technical Requirements

There are three technical requirements for the Figure CFD application.

Figure CFD should be an application running on both Android and iOS tablets. It shall be written in a framework called React Native to be better maintainable for the development team because all of the other mobile applications are written in React Native as well.

Figure CFD shall be integrated with the application Figure POS. Synchronization between these two applications should be fast and work even during an internet outage. Therefore, these applications should communicate with each other only on a local network.

Figure's development team shall be able to monitor application health in production. To make the application stable and provide good service to the merchants, Figure's team needs to know if the application is crashing or if the communication on the local network does not work so that they can act on it.

Functional Requirements

Based on the previous section, two actors, the merchant and customer, use the application Figure CFD. The use case diagram shows the base use cases for each actor. The diagram was used in the project's planning phase to scope it down.

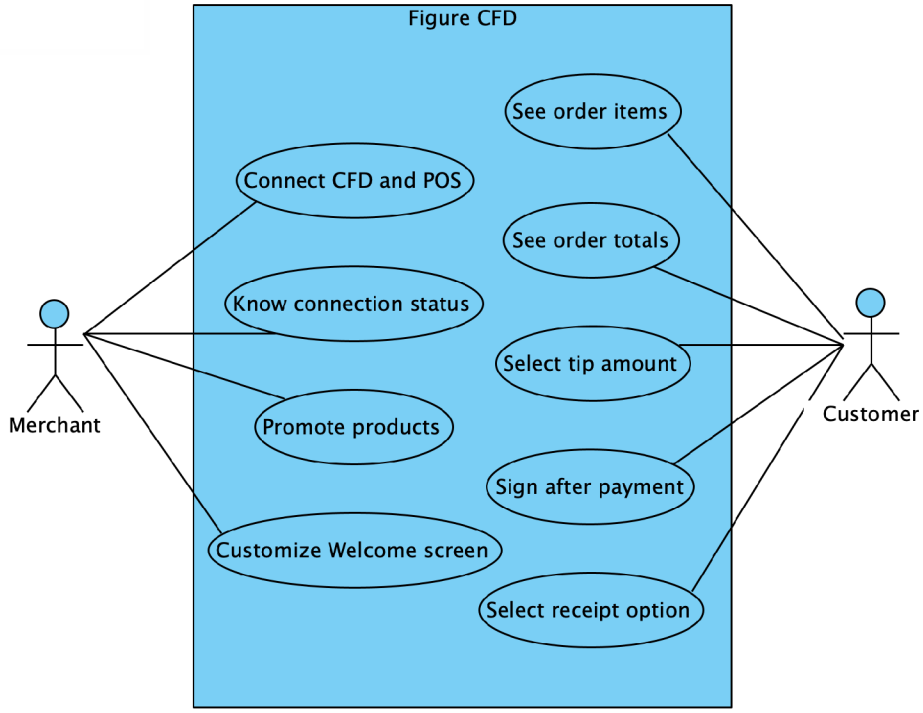

Figure 3.1: Figure CFD - Use Case Diagram

3.4 Local Network Communication

This section describes how the library React Native Local Server is utilized to accomplish synchronization between applications Figure CFD and Figure POS.

Because the primary goal of the communication is to synchronize the state of the two applications, we decided to leverage the Messaging functionality of the library. It also brings the benefit of having the Zeroconf working without significant effort. However, it is essential to note that the Zeroconf layer was added to the library later when the Figure CFD project was already in progress.

Client-Server Model

When designing the communication between Figure CFD and Figure POS, a significant decision was which application should be considered as a server and which as a client. One of the benefits of considering the POS a server was that one POS could control multiple CFDs at a time. On the opposite side, making the CFD a server would mean that the POS connects to the CFD and not vice versa. The final decision was to make Figure CFD a server. The main reason was that CFD should be considered an external accessory as a payment terminal. Therefore, it makes sense that the POS connects to CFD and not the other way around. The benefits of one-to-many communication were ignored because there was no business use case.

Zero-configuration

The zero-configuration concept was introduced to the project after some merchants were already testing the MVP. The main goal of adding a zero-configuration was to ease the setup and make the connection more stable.

Before having the zero-configuration, the users of POS had to either scan the QR code displayed on CFD or input the IP address manually to connect both apps. This was fine during the setup, but once both screens were physically installed it was causing issues. There was one more technical issue. If the connection between POS and CFD got interrupted, POS had to retry connecting back to the CFD periodically.

After utilizing the zero-configuration functionality provided by the library React Native Local Server, the users can see all available CFDs on their network. They do not have to manually scan any QR code or input the IP address. Also, POS does not have to retry the connection on the interruption. Thanks to the zero-configuration functionality, the POS gets notified once the CFD is back online and ready to accept connections. Lastly, the CFD can run on a dynamic port instead of a static one.

Messages

The synchronization of Figure CFD and Figure POS is achieved through messaging functionality, where messages serve as the core data structures of communication. These messages fall into two groups.

The messages in the first group relate to the business configuration. They allow setting the promotion images and configuring the welcome screen, such as logo, welcome text, and background images.

The second group deals with the order process and can be divided into three categories based on the direction from the perspective of the companion application: outgoing, incoming, and bidirectional. The outgoing category includes messages that notify CFD about order updates and request signatures, tips, and receipt information. The incoming category contains responses to those requests. Furthermore, one message in the bidirectional group notifies the other application about changing the tip amount. For the full specification of all messages, please refer to Appendix B or the source code of the SDK. For an illustration of an example sequence of the messages, please see Figure 3.2.

Figure CFD SDK

One of the challenges when designing the communication between Figure CFD and Figure POS lies in sharing message definitions as well as the type definition of domain objects, such as order, welcome screen configuration, and more. We wanted to avoid duplicating the same code on both applications because it often ends in inconsistencies, thus, potential issues and even crashes.

The solution to this problem is a TypeScript package published on the npm registry. This package, called @figuredev/cfd-sdk, consists of all type definitions that are shared across the two applications and factory methods of messages to improve developer experience when integrating with the Figure CFD. The SDK also exports two RxJS operators that can be used in message handlers to filter only specific messages. Lastly, the SDK provides a factory method that simplifies instantiating the MessagingClient class and fills in some of the type generics to match the CFD use case. This package is then installed in both applications, and they can import any related type definitions.

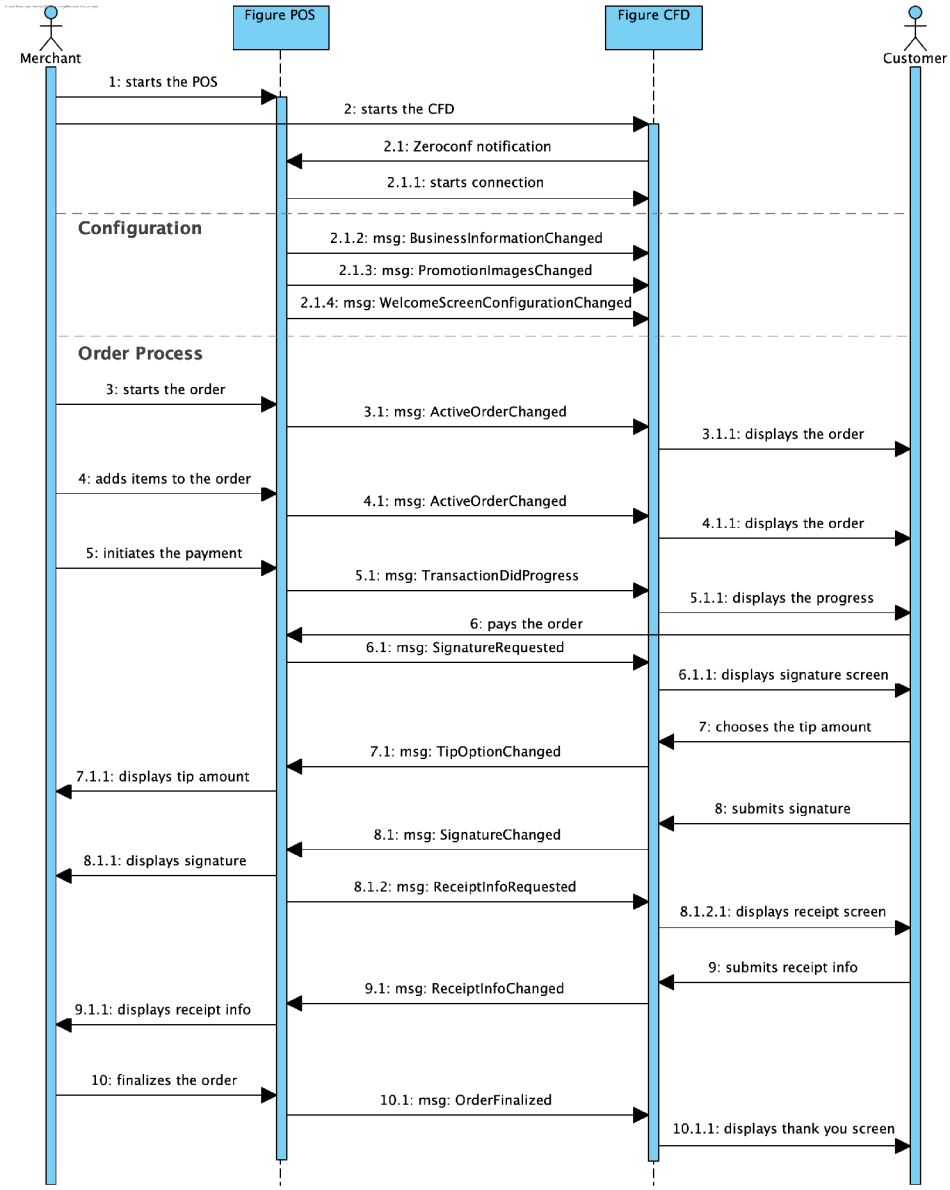

Figure 3.2: Figure CFD - Example Message Sequence

3.5 Navigation

Figure CFD consists of 8 screens and several popovers. This section describes how the navigation was implemented in the application Figure CFD. It presents each screen, its visuals, and its purpose. It also explains how the transition between each screen works because the navigation is not triggered by the user in most cases.

Navigation Tree

Navigation in Figure CFD was implemented with a third-party library called React Navigation. The library exposes three types of React components that allow for a declarative approach to navigation.

The first component is the NavigationContainer. It is a component that should be used in every application just once. This wrapper component should be mounted as closest to the top of the component tree as possible. It holds the navigation state of the application and changes it accordingly.

The second type of component is called the navigator. Three specific navigators are exposed by React Navigation: Stack Navigator, Drawer navigator, and Tab Navigator. These components should wrap up a group of screens on the same level. Using a specific navigator affects the behavior and visuals of the application.

The last type of component is called the screen. This component wraps every screen and popover in the application. Each navigator has its own screen component. The screen can render a navigator, which causes the whole navigation to be a tree.

The navigator on the top level of Figure CFD is a Stack Navigator. It has five screens for children: Setup screen, Welcome screen, Configuration screen, Active Order screen, and Thank You screen. The Configuration screen then renders a Drawer Navigator with two child screens. Lastly, the Active Order screen renders a second Stack Navigator with screens like Order Detail screen, Signature screen, and Receipt screen.

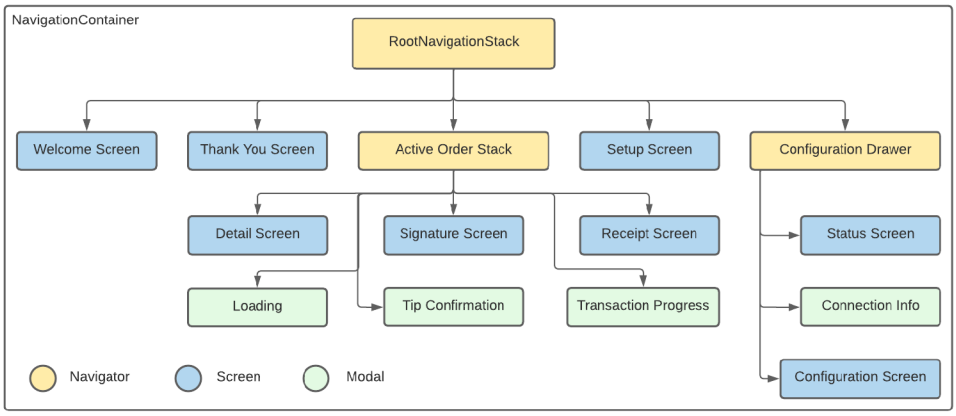

Figure 3.3: Figure CFD - Navigation Tree

Navigation Flow

In most applications, user actions, like pressing a navigation button, drive the navigation state. However, since Figure CFD is considered an external accessory, we decided that the navigation state will be driven mainly by the connected application, therefore, messages sent from it. There are four main states in which Figure CFD can be, and each splits into one or multiple screens.

Once the user starts the application for the first time, the application is in a setup state. If the setup was completed, the application can never reach this state again. While idle, the application displays a welcome screen from which users can navigate to the configuration state. The transfer to the active order state is driven by a message with a new order.

While the application is in the active order state, it can display three screens and several popovers. The first screen is the Order Detail screen. The order data displayed on this screen are updated with the same message type that initiated the state. While there is an active order, the connected application can change the navigation to the Signature screen by sending a message with this request and configuration for that screen. Another message type can be sent to cancel the signature and navigate back to the detail screen. In the same way, the connected application can invoke navigation to the Receipt screen. Moreover, in

addition to these screens, it can also force the CFD to display payment progress as a popover. There are two message types designed to exit the active order state. One of them navigates the CFD to the Welcome screen right away, and it should be used in cases when the order gets canceled. The other message tells the CFD to briefly display the Thank You screen and then navigate automatically to the Welcome screen.

For more information, see the following diagram (Figure 3.4) and screens description.

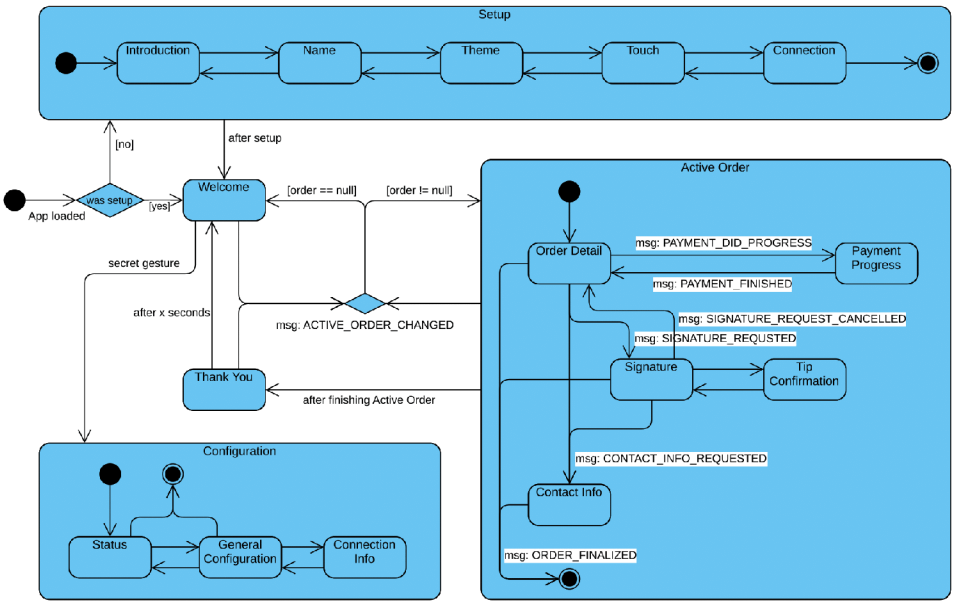

Figure 3.4: Figure CFD - Navigation Flow

Setup screen

The very first screen a user sees is the Setup screen. It is displayed just when the app starts for the first time. If the setup had been completed, this screen would never show again. The purpose of this screen is to explain what Figure CFD is, configure the basic settings, and teach the user how to use the app.

The user has to take five steps to complete the setup. Firstly he can see a text describing what Figure CFD is and what it does. In the second step, the user gives a name to this specific CFD. This name is later displayed in the POS application. This feature is useful mainly for businesses that plan to use more than one CFD. In the third step, the user can choose between dark and light modes. The mode change is performed immediately so the user can compare both modes. In the fourth step, the user is instructed on how to reach the configuration screen, which is hidden from the end customer in the application, and can only be accessed through a specific gesture. The last step is about connecting the CFD to a POS. The user is given instructions and a QR to scan from the POS. Throughout the whole setup, the users can see their progress.

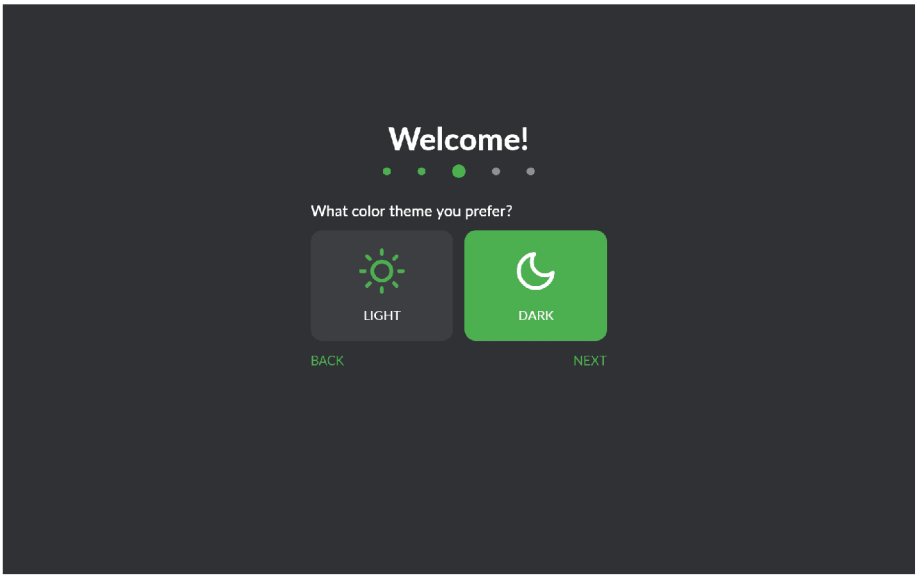

Figure 3.5: Figure CFD - Setup Screen

Welcome screen

While the CFD is in an idle state, meaning there is no active order, it displays the Welcome screen. This screen should attract the customer's eyes once they approaches the counter. Therefore it should be visually appealing.

The merchant can configure one or multiple images to be shown on the Welcome screen. If more than one image is provided, those images slide in a carousel. Besides the images, the merchant can configure three more parts of the Welcome screen. Those are the logo, business name, and welcome text. Each of them is optional and displayed in the middle of the screen in a white rectangle.

If there is some issue with the CFD, like no POS is connected, the customer can see an error message in the upper right corner to warn the merchant.

Currently, there is just one layout, which might only be convenient for some merchants. The problem with this layout is that it blocks the center of the screen, therefore, the center of the image. In the future, there might be a setting that will allow the merchant to customize the layout more.

Figure 3.6: Figure CFD - Welcome Screen

Order Detail screen

The screen that is displayed to the customer for the longest time and holds the most information is the Order Detail screen. It consists of three sections.

The most considerable section is on the left side and occupies the most space. It is an item breakdown section. The customer can see all the items in the order with their modifications and prices.

Then a section in the bottom right corner displays order totals. This section shows how much the customer is being charged and why. There is a breakdown that shows individual discounts, fees, and taxes. Also, if the final price differs based on the payment method, the customers can see how much they can save.

The last section is there to motivate the customer to buy more products. It is located in the upper right corner. The merchant can configure multiple promotional images shown in the carousel there. If the customer does not find the image interesting, they can swipe it to the left.

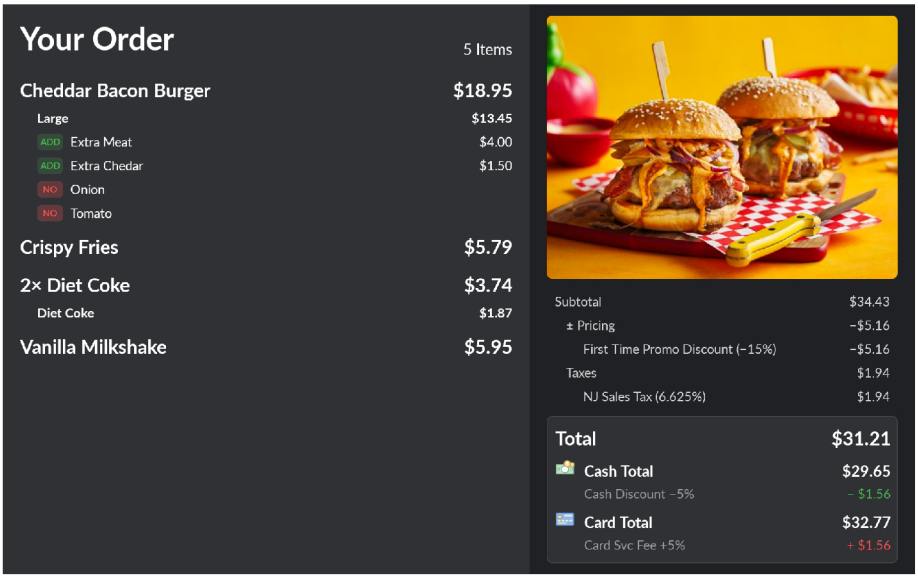

Figure 3.7: Figure CFD - Order Detail Screen

Signature screen

Since the application's primary market is the US, it must respect their common customs. The main difference between Europe and US that affects Figure CFD lies in tipping and card payment. While in Europe, we are used to tipping the server before the payment, in the US, it is the other way around. They process the payment, and after that, they input the tip, which is charged later. The second difference is that it is still common in the US to sign receipts to confirm card payments.

Figure CFD has one screen handling both steps because both occur after payment. The end customer can choose the tip amount from predefined values or input a custom amount. The selected tip option synchronizes with the connected application for better transparency. In addition to tipping, the customer has to sign the order before finishing the order. If they make a mistake while signing, they can always clear the signature and sign again. Once the customer inputs the tip and signature, they press the continue button.

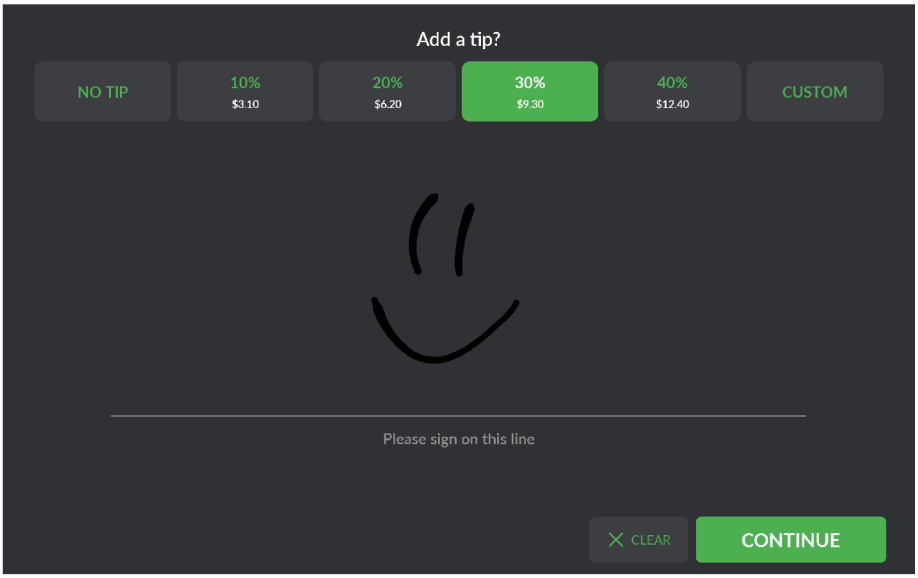

Figure 3.8: Figure CFD - Signature Screen

Receipt screen

During the checkout process, the customer can be prompted to choose a receipt type they would like to receive. Once that happens, the customer can see the Receipt screen. The purpose of this screen is to let customers choose what receipt they wants. Thanks to that, the merchant can save some money on paper, and the whole checkout process is faster because the customer can write down their contact information on their own.

There are three prominent buttons on the screen, each representing a type of receipt. The customer can select multiple receipt types or decide to get no receipt at all. If the customer chooses either email or SMS receipt, they are asked to fill in their email address or phone number respectively.

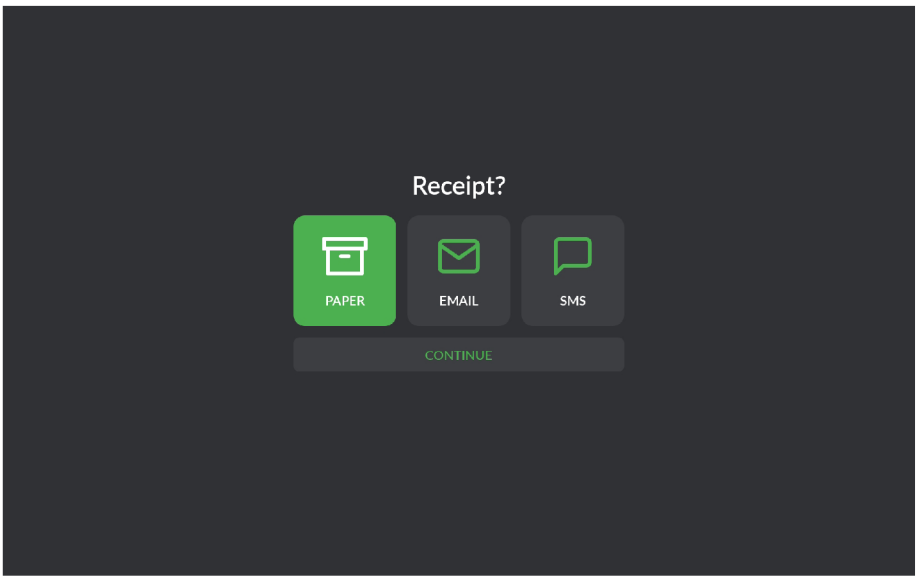

Figure 3.9: Figure CFD - Receipt Screen

Thank You screen

The very last screen shown to the customer during the checkout process is the Thank You screen. The purpose of this screen is to let the customer know that the whole ordering process is finalized and to motivate the customer to return to the restaurant later.

The current version displays just the text "Thank You!" on the Figure's primary green color. The text is animated to make the experience more pleasant. After a short period of time, the screen automatically changes to the Welcome screen to let other customers make an order.

Going forward, additional information will be displayed on this screen to the customer. For instance, if the customer were a member of the restaurant's loyalty program, they would be able to view the number of points they earned for that particular order.

Figure 3.10: Figure CFD - Thank You Screen

Configuration screens

There are two configuration screens whose purpose is to allow the restaurant to adjust the configuration of the CFD and troubleshoot connection issues. These screens are organized in a drawer navigator, and the users can access them from the welcome screen by performing the secret gesture.

The first screen shows the status of the CFD. Users can see if it is ready to accept new connections or what is the reason why it cannot. The screen also displays if there is any companion application connected to the CFD and its name. If the merchant needs to know the CFD's IP address and port number, they can also find it on this screen. Lastly, the screen provides information about the application, such as its version, and allows the users to check for updates.

The second screen in the drawer is a general setting screen. This screen shows information about the business provided by the companion application. It allows users to switch between dark and light modes. Lastly, there is also a section dedicated to development purposes, which allows to simulate application crashes and more. This section is visible only in the non-production builds of the application.

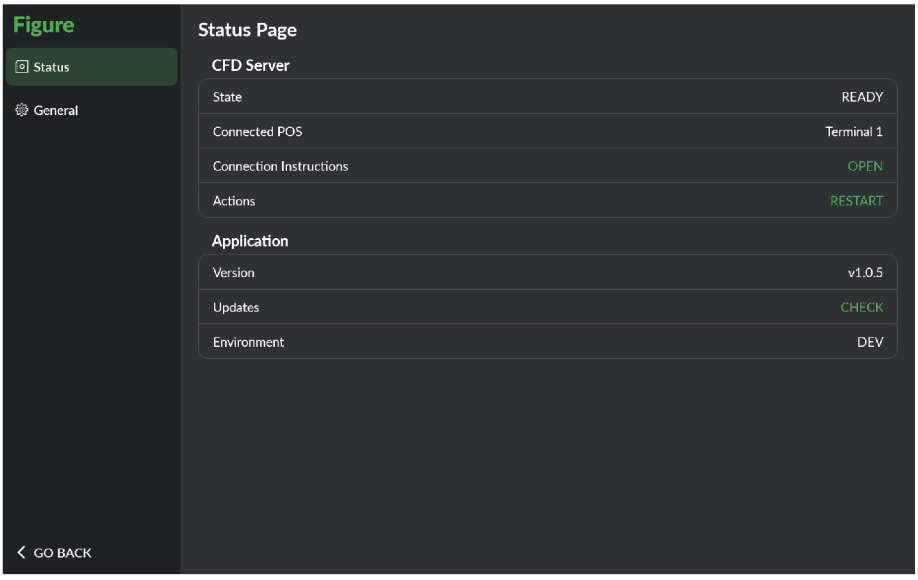

Figure 3.11: Figure CFD - Status Screen

3.6 Monitoring

Monitoring mobile applications is essential for ensuring their proper functioning and usability. Mobile applications can be affected by various issues, such as crashes, slow loading times, and bugs, that can significantly impact user experience. Effective monitoring of mobile applications can help identify and resolve these issues quickly before they cause user dissatisfaction and potential revenue loss. Additionally, monitoring can provide valuable insights into user behavior and usage patterns, allowing developers to optimize applications for better performance and engagement.

This section focuses on monitoring the application Figure CFD, its implementation, what valuable data it brings, and what the takeaways are. When designing the monitoring, there were three things to consider. The most basic but critical element is crash monitoring. Then there is usability monitoring, which provides insight into how the users utilize the application. Lastly, specifically for Figure CFD, we focused on monitoring local network communication stability.

To monitor application crashes, we decided to take advantage of the existing service called Bugsnag. Bugsnag provides React Native SDK, which, after setup, automatically sends crash reports to their platform. In order to make the reports human-readable, Bugsnag needs source maps and debug symbols to deobfuscate the source code. These files are automatically uploaded to the Bugsnag on every build from the CI system. All crashes are grouped and displayed in a dashboard. Bugsnag also provides a straightforward metric that indicates the application's stability. It is a percentage value representing how many active users have experienced a crash that day. The maintainers of the application then can set a target value. Currently, on the 30th of April 2023, the stability of Figure CFD is critical based on Bugsnag, see Figure 3.12. We have many crash reports related to memory issues, and the investigation is in progress.

Monitoring the application usage and stability of the communication is accomplished by integrating with a service called Datadog. The baseline lies in sending logs to their system, which help investigate issues reported by the users. Besides log management, Figure CFD leverages Datadog's Real User Monitoring, RUM, which provides insights into the application's use, such as button presses, navigation

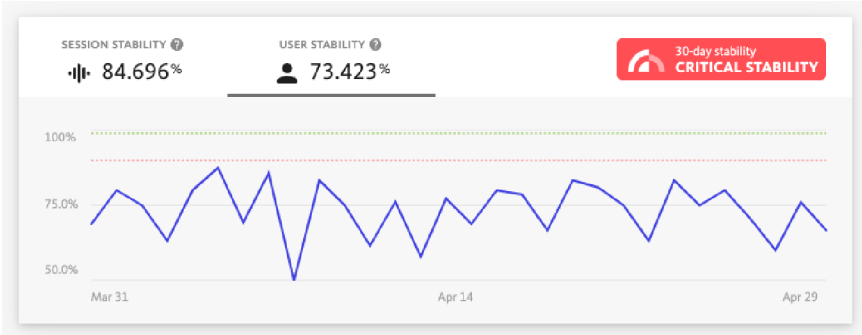

Figure 3.12: Figure CFD - Bugsnag Stability Report

transitions, and more. RUM data allows us to track version adoption, which helps us see how many CFDs are in production use and if any are running old versions. Based on these data, there are about 20 displays in use daily. Because the primary purpose of CFD is to display order details and improve the checkout process, we decided to track three business metrics. The first metric says how many unique orders were displayed, the second metric says how many of these orders were finalized, and lastly, how many orders were signed, which indicates mainly orders paid by card. In April 2023, Figure CFDs displayed about sixty thousand orders, making it two thousand daily. See Figure 3.13 for better insight.

The last key element of Figure CFD to monitor is the stability of local network communication. It is crucial to monitor because Figure CFD is the first production application that uses the library React Native Local Server. We decided to monitor two metrics, the number of disconnections with their reason and the time it takes to establish a new connection. It is essential to know these two values to asses how communication works overall. In April 2023, we tracked about fourteen thousand disconnections, most caused by the companion application closing the connection. These are under further investigation because our monitoring of the CFD does not provide additional information. Luckily, the disconnections are not severe because, in 75% of instances, a new connection was established within two seconds, causing only minor interruptions to the restaurant. For a more precise understanding, refer to Figure 3.14.

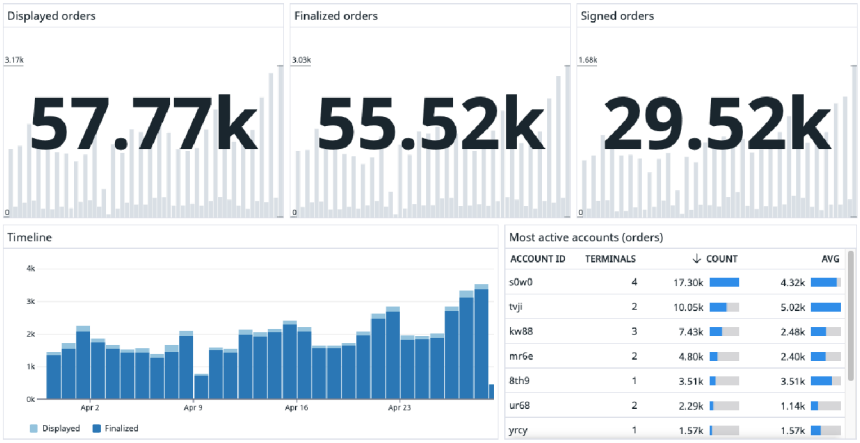

Figure 3.13: Figure CFD - Datadog Business Monitoring

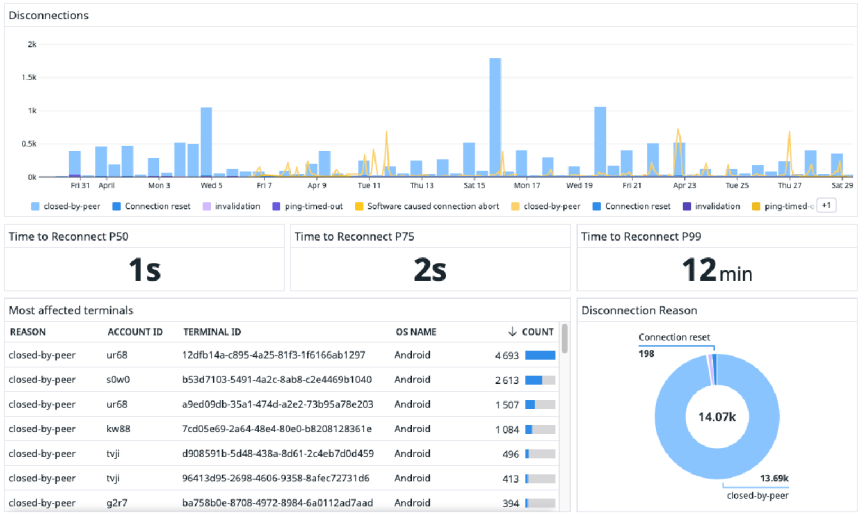

Figure 3.14: Figure CFD - Datadog Communication Monitoring

Conclusion

This thesis aimed to introduce a new React Native library that should provide an interface allowing developers to implement local network communication between their applications.

The library is called React Native Local Server and is available on GitHub and published on npm. The library provides multiple packages to its users. Three low-level abstraction packages implement the TCP, UDP, and Zeroconf protocols. Besides these, the library also exposes a package with a higher level of abstraction, allowing programming the communication between applications declaratively.

The second goal and motivation behind introducing the library were to implement an application for US company Figure, Inc. The application is called Figure CFD, and it optimizes the checkout process in quick-service restaurants. The application utilizes the library to synchronize with another React Native application to provide the best user experience possible. Figure CFD currently runs on around twenty devices a day.

Even though the library and the application suffer from some stability issues, they are both actively used in the production environment daily, and those stability issues are being investigated. After resolving those issues, new features are pending for the application, Figure CFD, such as displaying loyalty program information to the end customers.

Bibliography

1. *Native Modules Intro* [online]. Meta Platforms, Inc., 2023 [visited on 2023-02-05]. Available from: https://reactnative.dev/docs/native-modules-intro.
2. *What is a bridge in React Native?* [online]. Geeks for Geeks, 2022 [visited on 2023-02-05]. Available from: https://www.geeksforgeeks.org/what-is-a-bridge-in-react-native/.
3. *React Native's upcoming re-architecture* [online]. collective mind development d.o.o., 2021-10-08 [visited on 2023-02-15]. Available from: https://collectivemind.dev/blog/react-native-re-architecture.
4. KUROSE, J. F.; ROSS, K. W. *Computer networking: A top-down approach*. 6th ed. Pearson, 2012. ISBN 0-13-285620-4.
5. STEINBERG, Daniel H.; CHESHIRE, Stuart. *Zero Configuration Networking: The Definitive Guide*. O'Reilly Media, Inc., 2005. ISBN 9780596101008.
6. DAINEALS, Paul P.; ATENCIO, Luis. *RxJS in Action*. Manning Publications Co., 2017. ISBN 9781617293412.
7. *Epics* [online]. [visited on 2023-02-05]. Available from: https://redux-observable.js.org/docs/basics/Epics.html.
8. BARTLEY, Kevin. *What Is Caller ID and How Does It Work?* [online]. OnSIP, 2022 [visited on 2023-04-22]. Available from: https://www.onsip.com/voip-resources/voip-fundamentals/how-caller-id-works-common-questions-and-answers.
9. *Whozz Calling? Basic POS: For use with analog phone lines* [online]. [visited on 2023-04-22]. Available from: https://callerid.com/basic_unit.php?tab=integration.
10. TEJEDA, Vivian. *Customer Facing Displays: Benefits and How To Set Them Up* [online]. 2022-05-23. [visited on 2023-04-22]. Available from: https://www.shopify.com/retail/customer-facing-display.

A List of Digital Attachments

This thesis is submitted with following digital attachments:

- react-native-local-server.zip - source code of the library React Native Local Server. This library is also available on GitHub at `https://github.com/FigurePOS/react-native-local-server`.
- figure-cfd.zip - source code of the application Figure CFD. It also contains source code of the Figure CFD SDK, which is located in the sdk directory.
- demo.mp4 - short video footage of the application Figure CFD in action.

B List of Figure CFD Messages

Configuration Messages

BusinessInformationChanged

This message should be fired right after successful connection or when business information changes.

```
type BusinessInformationChangedMessage = {
    type: CFDMessageType.BusinessInformationChanged
    payload: {
        info: BusinessInformation | null
    }
}
```

PromotionImagesChanged

This message should be fired when promotion images change. Promotion images are displayed on the Order detail screen.

```
type PromotionImagesChangedMessage = {
    type: CFDMessageType.PromotionImagesChanged
    payload: {
        images: ImageData[]
    }
}
```

WelcomeScreenConfigurationChanged

This message should be fired when welcome screen configuration changes.

```
type WelcomeScreenConfigurationChangedMessage = {
    type: CFDMessageType.WelcomeScreenConfigurationChanged
    payload: {
        configuration: WelcomeScreenConfiguration | null
    }
}
```

Order Messages

Outgoing

ActiveOrderChanged

This message should be fired when active order changes.

```
type ActiveOrderChangedMessage = {
    type: CFDMessageType.ActiveOrderChanged
    payload: {
        order: Order | null
    }
}
```

SignatureRequested

This message opens Signature Screen on the CFD. The order id in payload must match the id of active order.

```
type ActiveOrderSignatureRequestedMessage = {
    type: CFDMessageType.SignatureRequested
    payload: {
        orderId: string
        signature?: SignatureConfiguration
        tip?: TipConfiguration
    }
}
```

SignatureRequestCancelled

This message closes Signature screen on the CFD and goes back to Order Detail. It can be used for example when the payment is voided.

```
type ActiveOrderSignatureRequestCancelledMessage = {
    type: CFDMessageType.SignatureRequestCancelled
    payload: {
        orderId: string
    }
}
```

TransactionDidProgress

This message should be on every step of the payment process. When CFD receives this message it opens / updates transaction progress popover.

```
type TransactionDidProgressMessage = {
    type: CFDMessageType.TransactionDidProgress
    payload: {
        orderId: string
        progress: TransactionProgress
    }
}
```

ReceiptInfoRequested

This message opens Receipt Info screen on the CFD.

```
type ActiveOrderReceiptInfoRequestedMessage = {
    type: CFDMessageType.ReceiptInfoRequested
    payload: {
        orderId: string
    }
}
```

ReceiptInfoRequestCancelled

This message closes Receipt Info screen on the CFD and goes back to Order Detail.

```
type ActiveOrderReceiptInfoRequestCancelledMessage = {
    type: CFDMessageType.ReceiptInfoRequestCancelled
    payload: {
        orderId: string
    }
}
```

OrderFinalized

This message should be fired as the last event for the order, because it displays Thank You screen and then navigates to Welcome screen.

```
type OrderFinalizedMessage = {
    type: CFDMessageType.OrderFinalized
    payload: {
        orderId: string
    }
}
```

Incoming

SignatureChanged

This message is fired from the CFD when customer inputs the signature. The signature property of payload is Base64 encoded PNG of the signature.

```
type ActiveOrderSignatureChangedMessage = {
    type: CFDMessageType.SignatureChanged
    payload: {
        orderId: string
        signature?: string | null
    }
}
```

ReceiptInfoChanged

This message is fired from the CFD when customer inputs the receipt info.

```
type ActiveOrderReceiptInfoChangedMessage = {
    type: CFDMessageType.ReceiptInfoChanged
    payload: {
        orderId: string
        receiptInfo: ReceiptInfo[]
    }
}
```

Bidirectional

ReceiptInfoChanged

This message is sent from the CFD, when the customer selects a tip option. It should be also sent from the POS, when the server changes the tip option.

```
type ActiveOrderTipOptionChangedMessage = {
    type: CFDMessageType.TipOptionChanged
    payload: {
        orderId: string
        key?: TipOption["key"] | null
        value?: TipOption["value"] | null
    }
}
```