SwiftShare

A Cross-platform File Management Application, Software Design Document.

Bautista, Ronald Francis D.
Binwag, Louis G. III
Cacacho, Jean Maximus C.
Magtipon, Ciana Louisse G.
Ozo, Kyle Joshua A.
Zabala, Paco Antonio V.

TABLE OF CONTENTS

1. INTRODUCTION

1.1. Purpose

The purpose of this **Software Design Document** is to provide a comprehensive description of the architecture, components, and design specifications for SwiftShare, a cross-platform file transfer application. This document serves as a reference for software developers, designers, and stakeholders (clients), ensuring that all technical aspects of the system are well-defined before development begins.

The primary objective of SwiftShare is to **enable seamless, offline file transfers** between Windows, macOS, Linux, Android, and iOS devices using LAN (Wi-Fi) and Bluetooth, without relying on cloud storage or internet connectivity.

This document includes the following information:

System Overview	A high-level description of how the system operates.
Systems Architecture	The structural design and division of responsibilities into various subsystems.
Component Design	Breakdown of key components and their ideal system interactions.
Data Design	Methods of file handling, transferring, networking, to ensure seamless file management through different devices.
Human Interface Design	User interface (UI) elements and user experience (UX) considerations.
Requirements Matrix	A mapping of functional requirements to system components.

This document was created to ensure clarity and consistency through the project development.

1.2. Scope & Limitations

SwiftShare is a **cross-platform offline file transfer application** designed to enable **seamless file sharing between Windows, macOS, Linux, Android, and ~~iOS devices.~~** The application will support LAN-based transfers (Wi-Fi), and Bluetooth-based transfers without requiring an internet connection or cloud storage. The key features include:

Cross-Platform Support:	➔ Allows file transfers between Windows, macOS, Linux, and Android/iOS devices.
Multiple Transfer Modes:	➔ LAN-based file transfer using Java NIO for desktop devices. ➔ Bluetooth file transfer using BlueCove (desktop) and Android Bluetooth API (mobile).
User-Friendly Interface:	➔ Drag-and-drop functionality (JavaFX UI for desktop). ➔ Material Design UI for Android ➔ Progress indicators for real-time transfer tracking.
File Caching & History: **(if time permits)**	➔ Tracks recently transferred files for quick access.
Additional Features: **(if time permits)**	➔ File conversion (JPG to HEIC, Word to TXT, MOV to MP4). ➔ File compression (ZIP or other formats).

While SwiftShare aims to provide seamless file management among various operating systems, **the following limitations exists:**

- ➔ No Cloud / Internet Integration
- ➔ Bluetooth may have range limitations
- ➔ Local Area Network (LAN) / Wi-Fi Dependency
- ➔ iOS may be limited to LAN as there are security issues and certain native file system interactions may be restricted.
- ➔ File Size constraints may occur down the line. While its AirDrop counterpart is able to send large files, when used at *X* amounts cause pipelining issues.
- ➔ Limited Support for Legacy Devices: Devices running outdated operating systems (e.g., Windows 7, Android 5.0) may not be fully supported.

1.3. Overview

The Software Design Document is divided into 8 sections with various subsections. The sections of the Software Design Document are:

- Introduction
- System Overview
- System Architecture
- Data Design
- Component Design
- Human Interface Design
- Requirements Matrix
- Appendices

1.4. Reference Material

As the development of the project has not yet started, the group currently lacks specific resources and documentation. However, the team plans to utilize various Java libraries, including **Java NIO, BlueCove, Android Bluetooth API, and the Android SDK** for the Windows-Android implementation. Furthermore, the group intends to utilize **JavaFX** for the main graphical user interface to ensure cross-compatibility for the different operating systems that will be involved in the project development (Mac-Windows-Linux-Android Systems)

Current Documentations.

java.nio	https://docs.oracle.com/javase/8/docs/api/java/nio/package-summary.html
BlueCove	https://code.google.com/archive/p/bluecove/wikis/Documentation.wiki
Android Bluetooth API	https://developer.android.com/reference/android/bluetooth/package-summary
<insert name>	<insert link>

As the development of the project progresses, new documentations and reference material may be added to this document.

1.5. Definitions and Acronyms

Listed below are the various definitions and acronyms used throughout the project.

JavaFX.	A modern GUI framework for Java that allows the creation of interactive and visually appealing desktop applications.
Figma.	A cloud-based UI/UX design tool used for prototyping and designing the user interface of the application.
Java.	An object-oriented programming language that boasts cross-platform compatibility, making it ideal for our development of a multi-platform file management application.
JavaNIO.	A Java API for non-blocking I/O operations, which would be relevant in enabling efficient file transfers by utilizing buffers instead of direct stream-based I/O
Android SDK.	General collection of development tools and libraries that are relevant when working with android applications. This includes building, testing, and debugging applications.
Android-Bluetooth API.	A framework within the Android SDK that enables Bluetooth communication between Android devices for secure peer-to-peer file transfers.
Maven.	A build and dependency management tool for most Java Projects. This is to ensure all our systems have appropriate dependencies and mitigate back-end errors.
BlueCove.	A Java library that provides Bluetooth stack access on desktop platforms, enabling wireless communication via Bluetooth.
Insert	Insert

2. SYSTEM OVERVIEW

Offline GUI file transfer applications that are often restricted to specific operating systems or ecosystems, as seen with Apple's AirDrop. *While AirDrop offers a seamless and user-friendly experience, it is exclusively available for macOS and iOS devices,* limiting its usability across different platforms. Our project seeks to address this limitation by creating a *device-agnostic* offline file transfer application. By leveraging *Java's cross-platform capabilities*, we aim to provide a universal solution that enables efficient file sharing across diverse operating systems, ensuring accessibility and ease of use for all users.

3. SYSTEM ARCHITECTURE

3.1. Architectural Design

SwiftShare

The architectural design of SwiftShare defines the program structure and how different components interact to achieve seamless cross-platform file sharing. The application follows a modular architecture, dividing responsibilities into distinct subsystems to enhance maintainability, scalability, and efficiency. In order to create a more efficient application, the team decided to create a modular architecture such that each feature does not necessarily depend on the other, especially since there are different environments defined in the various operating systems involved.

3.1.1. SwiftShare Systems Architecture

The overview of SwiftShare subsystems are as follows:

A. **File Management**

- a. **File Transfer**
  - *i. Personal Computer (PC)*
    1. *Windows to Windows*
    2. *Linux to Linux*
    3. *Windows to Linux (vice versa)*
    4. *Windows to MacOS (vice versa)*
    5. *Linux to MacOS (vice versa)*
  - *ii. Mobile*
    1. *Android to Android*
  - *iii. PC to Mobile (vice versa)*
- b. **File Accessibility**
- c. **File Conversion (IF TIME PERMITS)**
  - *i. HEIC to JPEG*
  - *ii. Word to TXT*
  - *iii. MOV to MP4 (Tentative)*
  - *iv. <insert other possible file conversions>*
- d. **File Compression (IF TIME PERMITS)**
  - *i. ZIP*
  - *ii. RAR*
  - *iii. <insert other extensions for compression>*

Bluetooth Connection Protocol

Our program aims to support Bluetooth file sharing for small-medium sized files across desktop and mobile devices. This feature enables users to transfer files wirelessly without requiring an internet connection or a local area network.

The platforms that will support this feature are:

1. Windows 10/11
2. macOS 15
3. Debian-based Linux distributions (e.g. Debian, Ubuntu, Linux Mint etc.)
4. Android 15

~~5. iOS 18~~

As of writing these are the prevalent operating systems being run on most consumer computing devices. The choice to primarily support these platforms comes from the abundance of already existing developer tools for them which allows us to leverage these frameworks instead of building our own ad-hoc networking stack from scratch.

The bluetooth file-sharing feature is designed to prioritize:

1. **Ease of Use**
   The outputs of this subsystem should seamlessly integrate into the various front-ends (JavaFX, Android SDK, etc.) of the program, enabling the creation of good UI/UX.
2. **Cross-Platform Compatibility**
   Connections and therefore files should be able to be sent both to and from any of the operating systems listed above.
3. **Offline Accessibility**
   Internet or LAN will not be required for file transfer using this feature.

However, due to Bluetooth's bandwidth limitations, this feature is best suited for files under 100 MB. For larger files, the program's LAN-based implementation serves a more appropriate alternative as it can support higher transfer speeds and better reliability.

Local Area Network Protocol

The program aims to support file transfers between desktop and mobile devices connected to the same network. The program will establish a direct connection with another, allowing for efficient peer-to-peer file transfer.

The receiving device will designate itself as the server/receiver, and will wait for the client/sender to facilitate a request via TCP. After the handshake, the sender will be able to send a file to the receiver.

This feature will be supported by the following platforms:

1. Windows 7, 10, 11
2. Linux (Ubuntu, Debian)
3. macOS
4. Android 15

By leveraging Java NIO's capabilities, LAN file transfer would be able to handle the transfer of relatively larger files, as it is a non-blocking framework with functions that allow direct file reading/writing from the disk to memory.

3.1.2. Bluetooth Systems Architecture

The Bluetooth file-sharing subsystem will follow a client-server architecture, where devices establish a connection using Bluetooth Classic or Bluetooth OBEX.

A. System Components & Roles

a. Client
   - ↳ The initiating device that requests a file transfer.

b. Server
   - ↳ The receiving device that accepts and processes the transfer request.

c. Transport layer
   - ↳ Bluetooth stack that handles communication.

B. Communication Flow

a. Device Discovery & Pairing
   - ↳ Client scans for available Bluetooth devices within range. Once a target device is found and selected, pairing is established via SSP (Secure Simple Pairing) or legacy PIN-based pairing.

b. Service Discovery
   - ↳ The client queries the server for supported Bluetooth profiles, OBEX protocol will be prioritized however if not available then the program will perform raw file transfer via RFCOMM.

c. File Transfer
   - ↳ Files will be transferred through chunks via push or pull mechanism when using OBEX. Custom protocols are implemented when using RFCOMM.

d. Error Handling & Reconnection
   - ↳ IF the connection drops, the system automatically retries based on a defined timeout mechanism. Users receive feedback on transfer status via the front-end UI.

C. Technology Stack

a. Bluetooth Communication Layers
   - ↳ Windows: BlueCove
   - ↳ macOS & Linux: BlueZ Bluetooth Stack via Java DBus API
   - ↳ Android: Android Bluetooth API

b. Bluetooth Protocols
   - ↳ RFCOMM: stream-based file transfer
   - ↳ OBEX: structured file exchange

c. Libraries
   - ↳ BlueCove. *BlueTooth library for Windows, not actively maintained anymore but should still work on a minimal scale.*
   - ↳ BlueZ Bluetooth Stack. *Still actively maintained Bluetooth library built on top of BlueCove primarily for Linux but can also be compiled for macOS.*
   - ↳ Java NIO. *For efficient file I/O operations.*

D. Platform Specific Considerations

a. Desktop (Windows/macOS/Linux)
   - ↳ Java runs on the JVM and communicates with the native Bluetooth stack present on the hardware.

b. Android
   - ↳ Bluetooth operations will be handled via the Android Bluetooth API

c. iOS
   - ↳ As of now, the group has decided to withhold any development onto the iOS environment due to heavy restrictions by the Apple security system.

3.1.3. Local Area Network Systems Architecture

A. System Components & Roles

a. Server Component
   - ↳ The device listens for incoming connections on a specific port.
   - ↳ Server uses Java NIO's `ServerSocketChannel` to accept connections.
   - ↳ Following the establishment of the connection, it will read incoming file data into a `ByteBuffer` and write it to the disk.

b. Client Component
   - ↳ The device establishes a TCP connection with the server/receiver device.
   - ↳ Client uses a `SocketChannel` to send file data.
   - ↳ The file is read in chunks and the data is written to an output stream in a non-blocking manner.

c. Data Transfer Mechanism
   - ↳ Uses a MappedByteBuffer for file reading/writing.
   - ↳ Executes zero-copy file transfer using `FileChannel.transferTo()` and `FileChannel.transferFrom()`.

B. Communication Flow

a. Establishment of TCP Connection
   - ↳ Client selects a file to transfer and specifies the target IP address and port. Client then establishes a TCP connection using `SocketChannel`.
   - ↳ Server device is notified of the connection and is prompted to accept.

b. File Transfer Process
   - ↳ Client reads the file in chunks using `FileChannel` and writes to `SocketChannel`.
   - ↳ Server reads data from the socket then saves it on disk.
   - ↳ Server notifies the client that the transfer is completed.

c. Error Handling
    - ↳ In the case of unresponsive clients or connection interruption, the devices are informed of the error and the client is automatically disconnected.

C. Technology Stack

a. Libraries
    - ↳ android: Java library for Android API.
    - ↳ java.io: Java's standard input/output library.
    - ↳ java.net: Java's standard networking library.
    - ↳ java.nio: a more efficient alternative to Java's standard I/O and networking libraries, with functions for non-blocking operations.

b. Network Protocols
    - ↳ TCP/IP Protocol: communications standard that allows transmission of data between network devices.

D. Platform Specific Considerations

a. Android
    - ↳ Explicit permission is required for network operations and file reading/writing (which will be set up in *AndroidManifest.xml*).
    - ↳ Android enforces scoped storage, limiting app access to external storage. Files would need to be written to a specific folder.
    - ↳ Android does not allow network operations on the main UI thread. A separate thread must be instantiated for network operations.

3.1.4. Implementation Details

On Bluetooth Connectivity

Windows 10/11

For desktop computers running Windows 10/11 the Bluetooth communication layer used will be implemented with the BlueCove library along with the auxiliary OBEX library.

1. **Discovering Bluetooth Devices**

Before sending or receiving files, Bluetooth connections must first be detected and be able to be connected to. At a basic level we'll be implementing a `BluetoothDiscovery` class to handle this.

```
Java
import javax.bluetooth.*;

public class BluetoothDiscovery {
	public static void main(String[] args) throws BluetoothStateException {
	LocalDevice localDevice = LocalDevice.getLocalDevice();
	DiscoveryAgent agent = localDevice.getDiscoveryAgent();

	System.out.println("Local Bluetooth Device: " + localDevice.getFriendlyName());

	agent.startInquiry(DiscoveryAgent.GIAC, new DiscoveryListener() {
		public void deviceDiscovered(RemoteDevice btDevice, DeviceClass cod) {
		try {
			System.out.println("Found device: " +
btDevice.getFriendlyName(false) + " - " + btDevice.getBluetoothAddress());
		} catch (Exception e) {
			e.printStackTrace();
		}
		}
		public void inquiryCompleted(int discType) {
		System.out.println("Discovery Completed.");
		}
		public void servicesDiscovered(int transID, ServiceRecord[] servRecord) {}
		public void serviceSearchCompleted(int transID, int respCode) {}
	});
	}
}
```

The code above scans for nearby Bluetooth devices and prints their names and MAC addresses.

2. Pairing and Connection

The basic implementation of the `BluetoothConnector` class is as follows.

```java
import org.freedesktop.dbus.DBusConnection;
import org.freedesktop.dbus.exceptions.DBusException;
import org.freedesktop.dbus.interfaces.Properties;
import org.freedesktop.dbus.types.Variant;

public class BluetoothConnector {
       public static void main(String[] args) {
       try {
              DBusConnection conn =
DBusConnection.getConnection(DBusConnection.SYSTEM);
              String devicePath = "/org/bluez/hci0/dev_XX_XX_XX_XX_XX_XX"; //
Replace with actual MAC

              Properties deviceProps = conn.getRemoteObject("org.bluez",
devicePath, Properties.class);

              deviceProps.Set("org.bluez.Device1", "Paired", new
Variant<>(true));
              System.out.println("Device paired successfully!");

              deviceProps.Set("org.bluez.Device1", "Connected", new
Variant<>(true));
              System.out.println("Device connected!");

              conn.disconnect();
       } catch (DBusException e) {
              e.printStackTrace();
       }
       }
}
```

The code above outlines the connection process given an example MAC address.

3. Initializing File Transfer Session

BlueZ does not give direct control over RFCOMM sockets unlike BlueCove, as such we'll have to initialize our own OBEX session.

```
Java
import org.freedesktop.dbus.DBusConnection;
import org.freedesktop.dbus.DBusInterface;
import org.freedesktop.dbus.Path;
import org.freedesktop.dbus.Variant;
import java.util.HashMap;
import java.util.Map;

public class BluetoothObexSession {
       public static void main(String[] args) {
       try {
              DBusConnection conn =
DBusConnection.getConnection(DBusConnection.SYSTEM);
              DBusInterface obexClient = conn.getRemoteObject("org.bluez.obex",
"/org/bluez/obex", DBusInterface.class);

              // Define connection parameters
              Map<String, Variant<?>> args = new HashMap<>();
              args.put("Target", new Variant<>("OPP"));
              args.put("Destination", new Variant<>("XX:XX:XX:XX:XX:XX")); //
Replace with target MAC address

              Path session = (Path) obexClient.callMethod("CreateSession",
args);
              System.out.println("OBEX Session Created: " + session.getPath());

              conn.disconnect();
       } catch (Exception e) {
              e.printStackTrace();
              }
       }
}
```

4. Sending a File

We use the OBEX session we created above to implement file sending.

```java
import org.freedesktop.dbus.DBusConnection;
import org.freedesktop.dbus.Path;
import org.freedesktop.dbus.interfaces.DBusInterface;
import org.freedesktop.dbus.Variant;
import java.util.HashMap;
import java.util.Map;

public class BluetoothFileSender {
       public static void main(String[] args) {
       try {
              DBusConnection conn = DBusConnection.getConnection(DBusConnection.SYSTEM);
              DBusInterface obexClient = conn.getRemoteObject("org.bluez.obex",
"/org/bluez/obex", DBusInterface.class);


              Map<String, Variant<?>> sessionArgs = new HashMap<>();
              sessionArgs.put("Target", new Variant<>("OPP"));
              sessionArgs.put("Destination", new Variant<>("XX:XX:XX:XX:XX:XX")); //
Replace with target MAC


              Path session = (Path) obexClient.callMethod("CreateSession", sessionArgs);
              System.out.println("Session Path: " + session.getPath());

              Map<String, Variant<?>> transferArgs = new HashMap<>();
              transferArgs.put("Filename", new Variant<>("/home/user/sample.txt")); //
File to send

              Path transfer = (Path) obexClient.callMethod("SendFile", session,
transferArgs);
              System.out.println("File Transfer Started: " + transfer.getPath());

              conn.disconnect();
       } catch (Exception e) {
              e.printStackTrace();
       }
       }
}
```

The code above initializes an OBEX session to send a `.txt` file to the target device.

5. Receiving a File

To receive files we set up an OBEX server, following the proposed client-server architecture.

```java
Java
import org.freedesktop.dbus.*;
import org.freedesktop.dbus.interfaces.*;
import java.io.File;
import java.nio.file.Files;
import java.nio.file.StandardCopyOption;
import java.util.Map;

public class BluetoothFileReceiver {
        public static void main(String[] args) {
        try {
                DBusConnection conn = DBusConnection.getConnection(DBusConnection.SYSTEM);

                DBusInterface obexManager = conn.getRemoteObject("org.bluez.obex",
"/org/bluez/obex", DBusInterface.class);

                Path server = (Path) obexManager.callMethod("CreateServer", "opp");
                System.out.println("OBEX Server Started at: " + server.getPath());

                // Listen for incoming file transfers
                conn.addSigHandler(DBus.Signal.class, signal -> {
                if (signal.getName().equals("TransferRequested")) {
                        String transferPath = signal.getSource().getObjectPath();
                        System.out.println("Incoming transfer: " + transferPath);

                        // Accept the file transfer
                        DBusInterface transfer = conn.getRemoteObject("org.bluez.obex",
transferPath, DBusInterface.class);
                        transfer.callMethod("Accept");

                        // Wait for transfer to complete
                        while (true) {
                        Map<String, Variant<?>> properties = (Map<String, Variant<?>>)
transfer.callMethod("GetAll", "org.bluez.obex.Transfer1");
                        String state = properties.get("State").getValue().toString();
                        if (state.equals("completed")) {
                                System.out.println("File transfer completed!");

                                // Move the file from temporary storage
                                String filePath =
properties.get("Filename").getValue().toString();
                                File receivedFile = new File(filePath);
                                File targetFile = new File("/home/user/BluetoothReceived/" +
receivedFile.getName());
```

```
                              Files.move(receivedFile.toPath(), targetFile.toPath(),
StandardCopyOption.REPLACE_EXISTING);
                              System.out.println("File saved to: " +
targetFile.getAbsolutePath());
                              break;
                       }
                       }
               }
               });

               // Keep running to listen for transfers
               System.out.println("Listening for incoming Bluetooth file transfers...");
               Thread.sleep(Long.MAX_VALUE);

        } catch (Exception e) {
               e.printStackTrace();
        }
        }
}
```

The code above outlines how an OBEX server is initialized and receives files to save locally.

Android 15

As compared to the previous platforms the implementation for Android is relatively simpler as the API abstracts away protocol management through the `BluetoothSocket` class. Since Bluetooth workflows are much more common on mobile, Android has much more pre-built tools that can be used.

1. **Permissions & Setup**

   We first set up the proper device permissions to access Bluetooth functionality inside the configuration file.

Add to AndroidManifest.xml

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

       <!-- Declare required Bluetooth features -->
       <uses-feature android:name="android.hardware.bluetooth"
android:required="true" />
       <uses-feature android:name="android.hardware.bluetooth_le"
android:required="false" />

       <uses-permission android:name="android.permission.BLUETOOTH" />
       <uses-permission android:name="android.permission.BLUETOOTH_ADMIN" />

       <uses-permission android:name="android.permission.BLUETOOTH_CONNECT" />
       <uses-permission android:name="android.permission.BLUETOOTH_SCAN" />

       <!-- Storage permissions -->
       <uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"
/>
       <uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"
/>

       <application
       android:allowBackup="true"
       android:icon="@mipmap/ic_launcher"
       android:label="@string/app_name"
       android:roundIcon="@mipmap/ic_launcher_round"
       android:supportsRtl="true"
       android:theme="@style/Theme.BluetoothFileShare">
```

```
        <!-- Declare a foreground service for file transfer -->
        <service
              android:name=".BluetoothFileTransferService"
              android:enabled="true"
              android:exported="false" />



        <activity android:name=".MainActivity"
              android:exported="true">
              <intent-filter>
              <action android:name="android.intent.action.MAIN" />
              <category android:name="android.intent.category.LAUNCHER" />
              </intent-filter>
        </activity>

        </application>

</manifest>
```

2. **Discovering Bluetooth Devices**

The application starts by enabling Bluetooth functionality, listing all paired devices, and scanning for available connections. Unlike the other implementations, at this step devices are also already paired with.

```
Java
import android.Manifest;
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothManager;
import android.content.BroadcastReceiver;
import android.content.Context;
import android.content.Intent;
import android.content.IntentFilter;
import android.content.pm.PackageManager;
import android.util.Log;
import androidx.annotation.NonNull;
import androidx.appcompat.app.AppCompatActivity;
import android.os.Bundle;
import android.widget.Toast;
import java.util.Set;

public class BluetoothActivity extends AppCompatActivity {
       private BluetoothAdapter bluetoothAdapter;

       @Override
       protected void onCreate(Bundle savedInstanceState) {
       super.onCreate(savedInstanceState);

       BluetoothManager bluetoothManager = (BluetoothManager)
getSystemService(Context.BLUETOOTH_SERVICE);
       bluetoothAdapter = bluetoothManager.getAdapter();

       if (bluetoothAdapter == null) {
              Log.e("Bluetooth", "Device doesn't support Bluetooth");
              return;
       }

       if (!bluetoothAdapter.isEnabled()) {
              Intent enableBtIntent = new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);
              startActivityForResult(enableBtIntent, 1);
       } else {
              listPairedDevices();
              scanForNewDevices();
              }
       }
```

```
        private void listPairedDevices() {
        Set<BluetoothDevice> pairedDevices = bluetoothAdapter.getBondedDevices();
        for (BluetoothDevice device : pairedDevices) {
                Log.d("Bluetooth", "Paired Device: " + device.getName() + " - " +
device.getAddress());
        }
        }

        private void scanForNewDevices() {
        if (checkSelfPermission(Manifest.permission.BLUETOOTH_SCAN) ==
PackageManager.PERMISSION_GRANTED) {
                if (bluetoothAdapter.isDiscovering()) {
                bluetoothAdapter.cancelDiscovery();
                }
                bluetoothAdapter.startDiscovery();
                Log.d("Bluetooth", "Scanning for devices...");

                // Register BroadcastReceiver to handle new devices found
                IntentFilter filter = new IntentFilter(BluetoothDevice.ACTION_FOUND);
                registerReceiver(bluetoothReceiver, filter);
        } else {
                requestPermissions(new String[]{Manifest.permission.BLUETOOTH_SCAN}, 2);
        }
        }

        // BroadcastReceiver for discovering new devices
        private final BroadcastReceiver bluetoothReceiver = new BroadcastReceiver() {
        public void onReceive(Context context, Intent intent) {
                String action = intent.getAction();
                if (BluetoothDevice.ACTION_FOUND.equals(action)) {
                BluetoothDevice device =
intent.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);
                Log.d("Bluetooth", "New Device Found: " + device.getName() + " - " +
device.getAddress());
                }
        }
        };

        @Override
        protected void onDestroy() {
        super.onDestroy();
        unregisterReceiver(bluetoothReceiver);
        }
}
```

3. Send File

To send a file we open a `BluetoothSocket`, connected to a paired device, and start the data transfer,

```java
import android.bluetooth.BluetoothDevice;
import android.bluetooth.BluetoothSocket;
import java.io.File;
import java.io.FileInputStream;
import java.io.OutputStream;
import java.util.UUID;

public class BluetoothFileSender {
       private static final UUID OBEX_UUID =
UUID.fromString("00001105-0000-1000-8000-00805F9B34FB"); // OBEX Object Push UUID
       private BluetoothSocket socket;

       public void sendFile(BluetoothDevice device, File file) {
       try {
              socket = device.createRfcommSocketToServiceRecord(OBEX_UUID);
              socket.connect();

              OutputStream out = socket.getOutputStream();
              FileInputStream fis = new FileInputStream(file);
              byte[] buffer = new byte[1024];
              int bytesRead;

              while ((bytesRead = fis.read(buffer)) != -1) {
              out.write(buffer, 0, bytesRead);
              }

              fis.close();
              out.close();
              socket.close();

              Log.d("Bluetooth", "File sent successfully!");
       } catch (Exception e) {
              Log.e("Bluetooth", "Error sending file", e);
       }
       }
}
```

4. Receive File

To receive a file we use a `BluetoothServerSocket` to listen for incoming connections.

```java
Java
import android.bluetooth.BluetoothAdapter;
import android.bluetooth.BluetoothServerSocket;
import android.bluetooth.BluetoothSocket;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.util.UUID;

public class BluetoothFileReceiver {
        private static final UUID OBEX_UUID =
UUID.fromString("00001105-0000-1000-8000-00805F9B34FB"); // OBEX Object Push UUID
        private BluetoothServerSocket serverSocket;

        public void startServer() {
        try {
                serverSocket =
BluetoothAdapter.getDefaultAdapter().listenUsingRfcommWithServiceRecord("BluetoothFileRece
iver", OBEX_UUID);
                Log.d("Bluetooth", "Waiting for incoming connection...");

                BluetoothSocket socket = serverSocket.accept();
                InputStream in = socket.getInputStream();
                FileOutputStream fos = new
FileOutputStream("/storage/emulated/0/Download/received_file.txt");

                byte[] buffer = new byte[1024];
                int bytesRead;

                while ((bytesRead = in.read(buffer)) != -1) {
                fos.write(buffer, 0, bytesRead);
                }

                fos.close();
                in.close();
                socket.close();
                serverSocket.close();

                Log.d("Bluetooth", "File received successfully!");
        } catch (Exception e) {
                Log.e("Bluetooth", "Error receiving file", e);
        }
        }
}
```

The code above listens for connections and saves the file to local storage.

On Local Area Network Connectivity (LAN)
The following code represents the intended structure and functions of the feature. The code will be subject to change during the software development process.

I. Windows/Linux/MacOS
   A. Server
   1. The receiving device will serve as the server.
   2. Upon starting the file transfer process, the FileServer class will listen for incoming TCP connections. Upon establishing the connection, it will receive the file metadata, and then write the incoming file using `FileChannel.transferFrom()`.

```java
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class FileServer {
    private static int PORT; // port program listens to
    private static String SAVE_TO; // destination directory of file received

    public static void main(String[] args) {
        try {
            // create directory if it doesn't exist
            Files.createDirectories(Paths.get(SAVE_TO));

            ServerSocketChannel serverChannel = ServerSocketChannel.open();
            serverChannel.bind(new InetSocketAddress(PORT));
            System.out.println("Server listening on port " + PORT + "...");

            // wait for TCP connection
            while (true) {
                SocketChannel clientChannel = serverChannel.accept();
                new Thread(() -> handleClient(clientChannel)).start();
            }
        } catch (IOException e) {
            System.out.println("IOException occurred on client connection.");
        }
    }

    private static void handleClient(SocketChannel clientChannel) {
        try {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            clientChannel.read(buffer);
            buffer.flip();
            String metadata = new String(buffer.array(), 0, buffer.limit());
            String[] metaparts = metadata.split(";");
            String filename = metaparts[0];
            long fileSize = Long.parseLong(metaparts[1]);

            System.out.println("Receiving file: " + filename + " (" + fileSize
+ " bytes)");

            // save file to destination
            Path filePath = Paths.get(SAVE_TO, filename);
            try (FileChannel fileChannel = FileChannel.open(filePath,
StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
                long position = 0;
                while (position < fileSize) {
                    long transferred = fileChannel.transferFrom(clientChannel,
position, fileSize - position);
                    if (transferred <= 0) break;
                    position += transferred;
                }
            }

            System.out.println("File received: " + filename);
            clientChannel.close();
        } catch (IOException e) {
            System.out.println("IOException occurred on client handling.");
        }
    }
}
```

B. Client

1. The device sending the file will serve as the client.
2. The client will request a TCP connection to the server, send file metadata, then transfer the file using `FileChannel.transferTo()`.

Java
```java
import java.io.*;
import java.net.*;
import java.nio.*;
import java.nio.channels.*;
import java.nio.file.*;

public class FileClient {
    private static String SERVER_ADDRESS; // IP address of destination device
    private static int SERVER_PORT; // port number of destination device

    public static void main(String[] args) {

        // error handling for args
        if (args.length < 1) {
            System.out.println("Invalid args submission.");
            return;
        }

        // check if file exists
        String filePath = args[0];
        File file = new File(filePath);
        if (!file.exists() || !file.isFile()) {
            System.out.println("Invalid file.");
            return;
        }

        try (SocketChannel socketChannel = SocketChannel.open(new
InetSocketAddress(SERVER_ADDRESS, SERVER_PORT));
            FileChannel fileChannel = FileChannel.open(file.toPath(),
StandardOpenOption.READ)) {

            // send metadata
            String metadata = file.getName() + ";" + file.length();
            ByteBuffer buffer = ByteBuffer.wrap(metadata.getBytes());
            socketChannel.write(buffer);

            long position = 0;
            long fileSize = file.length();
            System.out.println("Sending file: " + file.getName() + " (" +
fileSize + " bytes)");

            // read out file
            while (position < fileSize) {
                long transferred = fileChannel.transferTo(position, fileSize -
position, socketChannel);
                if (transferred <= 0) break;
                position += transferred;
            }

            System.out.println("File sent successfully.");
        } catch (IOException e) {
            System.out.println("IOException on server connection.");
        }
    }
}
```

I. Android

A. Storage Permissions Setup

1. The following will be added to *AndroidManifest.xml* to allow network communication and file reading/writing.

```
Unset
<uses-permission android:name="android.permission.INTERNET"/>
<uses-permission android:name="android.permission.READ_EXTERNAL_STORAGE"/>
<uses-permission android:name="android.permission.WRITE_EXTERNAL_STORAGE"/>
```

B. Server

1. Upon starting the file transfer process, the FileServer class will listen for incoming TCP connections. Upon establishing the connection, it will receive the file metadata, and then write the incoming file using `FileChannel.transferFrom()`.
2. `FileServer_a` extends `AsyncTask` to enable running in a background thread using `doInbackGround()`.
3. Android API uses `Log` to print log output.

Java
```java
import android.os.AsyncTask;
import android.util.Log;
import java.io.File;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.ServerSocketChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.StandardOpenOption;

public class FileServer_a extends AsyncTask<Void, Void, String> {
    private static int PORT; // port program listens to
    private static final String SAVE_TO = "/storage/emulated/0/Download/"; //
save location

    @Override
    protected String doInBackground() {
        try {
            ServerSocketChannel serverChannel = ServerSocketChannel.open();
            serverChannel.bind(new InetSocketAddress(PORT));
            Log.d("FileServer", "Server listening on port " + PORT);

            // wait for TCP connection
            while (true) {
                SocketChannel clientChannel = serverChannel.accept();
                handleClient(clientChannel);
            }
        } catch (IOException e) {
            Log.e("FileServer", "Error: " + e.getMessage());
            return "Error receiving file.";
        }
    }
    private void handleClient(SocketChannel clientChannel) {
        try {
            ByteBuffer buffer = ByteBuffer.allocate(1024);
            clientChannel.read(buffer);
            buffer.flip();
            String metadata = new String(buffer.array(), 0, buffer.limit());
            String[] metaparts = metadata.split(";");
            String filename = metaparts[0];
            long fileSize = Long.parseLong(metaparts[1]);

            Log.d("FileServer", "Receiving file: " + filename + " (" + fileSize
+ " bytes)");

            // save file to destination
            File file = new File(SAVE_TO + filename);
            try (FileChannel fileChannel = FileChannel.open(file.toPath(),
StandardOpenOption.CREATE, StandardOpenOption.WRITE)) {
                long position = 0;
                while (position < fileSize) {
                    long transferred = fileChannel.transferFrom(clientChannel,
position, fileSize - position);
                    if (transferred <= 0) break;
                    position += transferred;
                }
            }

            Log.d("FileServer", "File received: " + filename);
            clientChannel.close();
        } catch (IOException e) {
            Log.e("FileServer", "Error handling client");
        }
    }
}
```

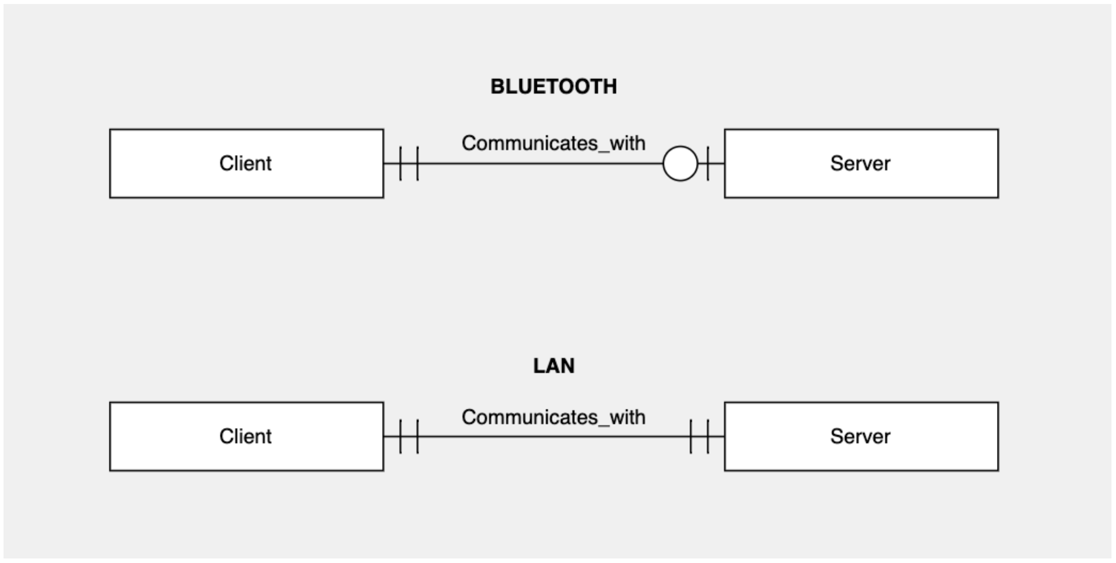
C. Client

1. The client will request a TCP connection to the server, send file metadata, then transfer the file using `FileChannel.transferTo()`.
2. As with the `FileServer_a`, `FileClient_a` extends `AsyncTask` so it can run in the background using `doInBackground()`.
3. Android API uses `Log` to print log output.

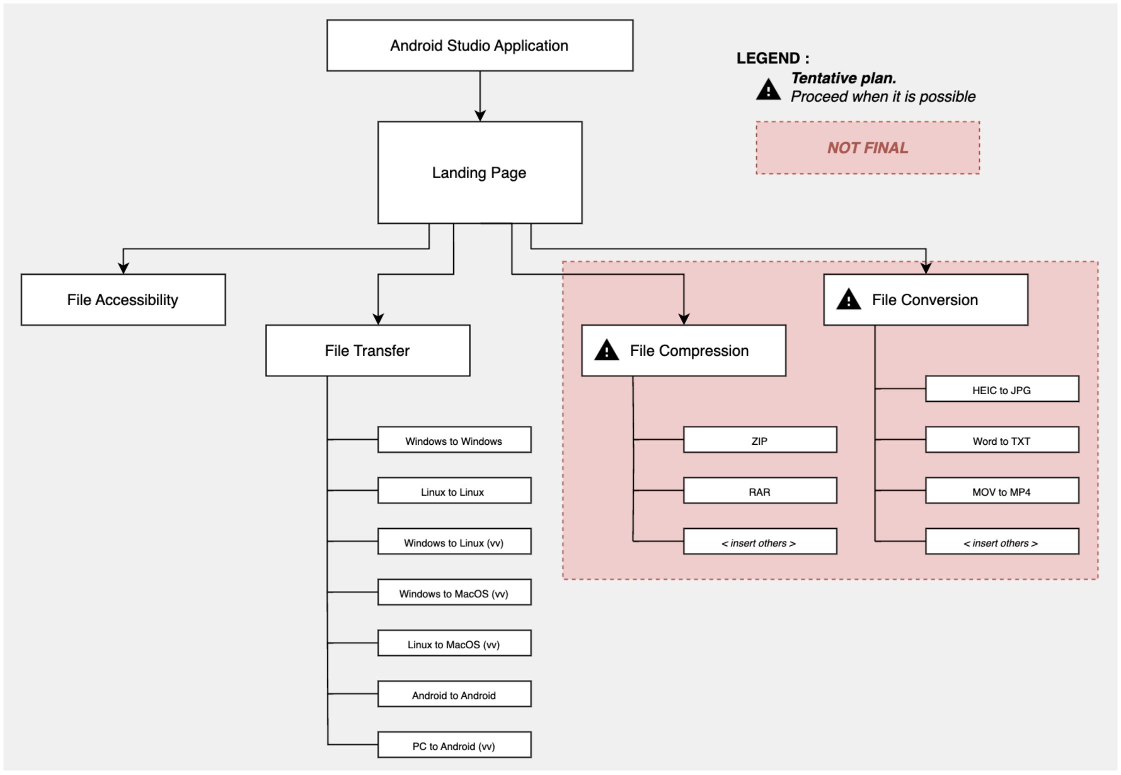
Java
```java
import android.os.AsyncTask;
import android.util.Log;
import java.io.File;
import java.io.IOException;
import java.net.InetSocketAddress;
import java.nio.ByteBuffer;
import java.nio.channels.FileChannel;
import java.nio.channels.SocketChannel;
import java.nio.file.StandardOpenOption;

public class FileClient_a extends AsyncTask<String, Void, String> {
    private static String SERVER_IP; // IP address of destination device
    private static int SERVER_PORT; // listening port of destination device

    @Override
    protected String doInBackground() {
        String filePath = params[0];
        File file = new File(filePath);

        if (!file.exists()) {
            return "File does not exist.";
        }

        try (SocketChannel socketChannel = SocketChannel.open(new
InetSocketAddress(SERVER_IP, SERVER_PORT));
            FileChannel fileChannel = FileChannel.open(file.toPath(),
StandardOpenOption.READ)) {

            // send metadata
            String metadata = file.getName() + ";" + file.length();
            ByteBuffer buffer = ByteBuffer.wrap(metadata.getBytes());
            socketChannel.write(buffer);
            buffer.clear();

            long position = 0;
            long fileSize = file.length();
            Log.d("FileClient", "Sending file: " + file.getName() + " (" +
fileSize + " bytes)");

            // read out file
            while (position < fileSize) {
                long transferred = fileChannel.transferTo(position, fileSize -
position, socketChannel);
                if (transferred <= 0) break;
                position += transferred;
            }

            return "File sent successfully.";
        } catch (IOException e) {
            Log.e("FileClient", "Error sending file.");
            return "Error sending file.";
        }
    }
}
```

3.2. Decomposition Description

Bluetooth and Lan Connectivity Entity Relationship Diagram

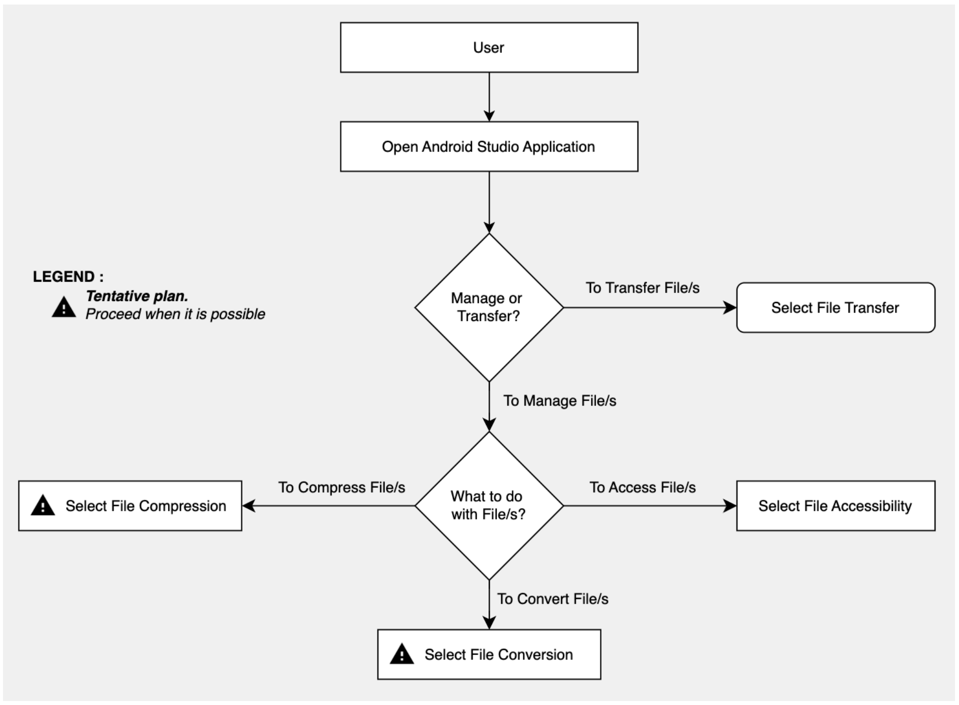


Information Architecture Diagram

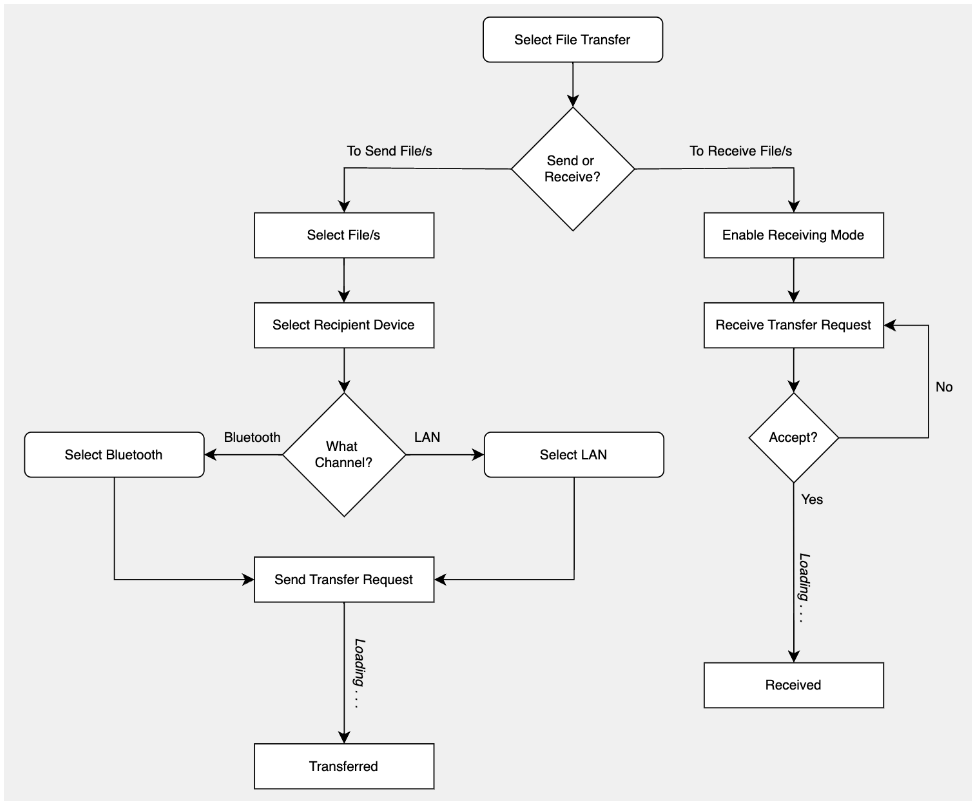


On Systems Data Flow Diagram

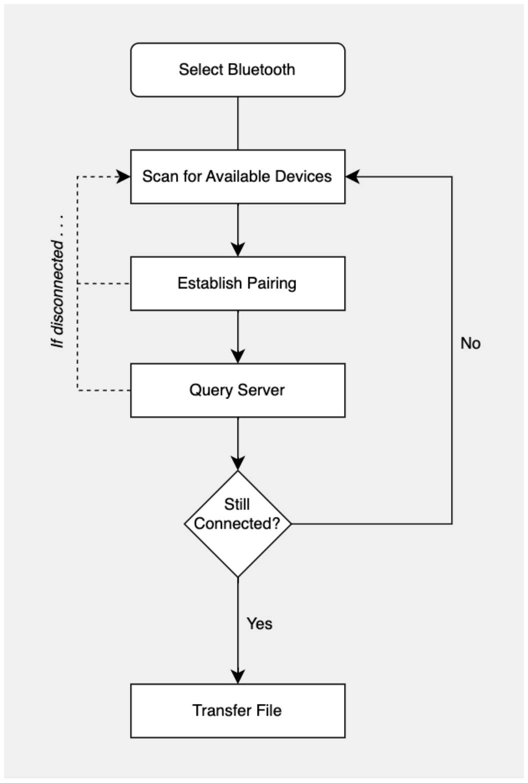
General Systems Architecture Data Flow Diagram



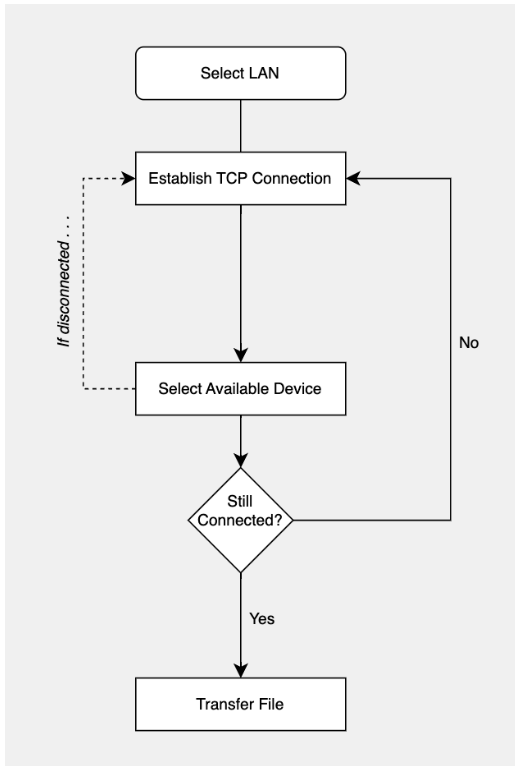
File Transfer Data Flow Diagram



On Bluetooth Systems



On LAN Systems



3.3. Design Rationale

SwiftShare was **designed with a cross-platform offline file-sharing application in mind**. With this, SwiftShare enables seamless file transfers between **Windows, macOS, Linux, and Android Devices, via Bluetooth and/or LAN-based connectivity.** The decision to adopt this modular architecture stems from a combination of usability, performance, maintenance, and efficiency.

SwiftShare follows a **modular architecture** to ensure a more efficient maintenance and across different operating systems. Each feature is encapsulated in independent modules, allowing seamless integration and upgrades without disrupting core functionalities.

High-Priority Modules:

- **File Management Module:** Handles file selection, access permissions, and storage organization. File System access via the application.
- **File Transfer Module:** Manages Bluetooth and LAN-based file transfers.
- **Device Discovery Module:** Enables device scanning and connection establishment.
- **User Interface Module:** Ensures a consistent UX across different platforms.
- **Data Storage Module:** Manages logs, history, and transfer records.

Medium-Priority Modules (If Time Permits):

- **File Conversion Module:** Allows format conversion (JPG to HEIC, Word to TXT, MOV to MP4, etc.).
- **File Compression Module:** Enables file compression for optimized transfers.

The modular structure ensures flexibility, making it easier to implement feature extensions in later stages of development. The decision to use **Java (Java NIO, BlueCove, BlueZ) and Android SDK** ensures versatile desktop and mobile compatibility.

However, due to complications with built-in device restrictions, we have decided to not include Apple iOS.

4. DATA DESIGN

4.1. Data Description

For SwiftShare on **PCs / Desktops**, the data storage will be dependent on where the user intends to store their files. It could be a central location such as `C:\Users\username\Documents\SwiftShareFiles`, or any desired location. However, SwiftShare will request access to relevant file locations of the users such as `Downloads, Documents, Desktop`, and other file locations that the user intends to involve in the file management application.

For **Android users**, it is standard practice to utilize the internal storage as default file location, and simply request access to relevant files such as `Internal Storage/Files`, or include an `Internal Storage/SwiftShare` folder. This is still subject to change as development progresses.

4.2. Data Dictionary

ENTITY	DESCRIPTION
Client	The initiating device that requests a file transfer in a client-server model.
Server	The receiving device that listens for file transfer requests and processes them.

5. COMPONENT DESIGN

The component design of SwiftShare organizes the system into distinct features ensuring its maintainability and efficiency. Each component interacts with others via defined interfaces to perform file transfer, device discovery (via Bluetooth/LAN), and internal storage management (logs, *caching(?)*, buffering, etc.)

1) **Sending a File**
    a) User selects a file → File Management Component
    b) User selects recipient device → Device Discovery & Connection Component
    c) User selects transfer mode (LAN or Bluetooth) → File Transfer Manager
    d) Transfer begins & UI updates progress → User Interface Component
    e) Transfer logs stored → Data Storage Component
2) **Receiving a File**
    a) User enables receiving mode → Device Discovery & Connection Component
    b) File transfer request received → File Transfer Manager
    c) User accepts/rejects transfer → User Interface Component
    d) File received and saved → File Management Component
    e) Transfer logs stored → Data Storage Component

The testing for this application will simply be through device connectivity, file transfer efficiency, and robustness of data transfers and event logging.

1.) Launching Application
2.) Allowing permissions (Bluetooth, LAN Discovery, Storage Access)
3.) Discovering Nearby Devices
4.) Connecting to Nearby Device (Sender to Receiver)
5.) Select File to Send
6.) Send File (Sender)
7.) Receive File (Receiver)
8.) Data Logging if Successful or Error Catching
9.) End

6. HUMAN INTERFACE DESIGN

6.1. Overview of User Interface

The Human Interface Design of SwiftShare is centered around **simplicity, efficiency, and cross-platform usability** to facilitate seamless offline file transfers across Windows, macOS, Linux, and Android Devices. The interface is designed to be **minimalistic yet feature-rich**, ensuring that users can easily navigate and perform file transfers without technical complexity.

6.2. JavaFX Implementation

JavaFX is a development framework that enables UI development on desktop platforms, namely Windows and Linux, for Java. With support for UI functionalities such as buttons, drag-and-drop, layouts, progress tracking, and user notifications, the JavaFX framework will greatly aid in UI development for desktop platforms. This project heavily utilizes JavaFX in UI development for desktop platforms for its ability to integrate with Java's networking APIs such as BlueCove, JavaBluetoothStack, and Java NIO. Through this, users will be able to access backend functionalities in a visually appealing and intuitive manner.

6.2.1. Class Descriptions

1. **Application Class**
    a. Application is a class in JavaFX from which JavaFX applications and projects extend in order to run.
2. **Scene Class**
    a. The Scene class acts as a container for all elements and UI objects in an application.
    b. Different scene objects can be created for pages in the application, such as scene objects for homepage, file selection, etc.
3. **Button Class**
    a. Button is a class in javafx.scene that is used to handle events in an application. It has three modes: normal (regular push button), default (receives a VK_ENTER keyboard press), and cancel (receives a VK_CANCEL keyboard press). Using an EventHandler object, a button can be used to handle events. In the application, a button connected to an EventHandler is useful as it can connect to backend processes, allowing for users to access them from the GUI.
4. **CheckBox Class**

   a. The CheckBox class in javafx.scene acts as a box that is filled with a check symbol when selected. It has three states: checked, unchecked, and undefined. These states can be accessed through the indeterminate and selected boolean properties. This is useful for selecting multiple files.

5. **ProgressBar Class**
   a. ProgressBar extends the class ProgressIndicator and displays progress in a horizontal bar, which can be set using the setProgress() method.

6. **Layouts**
   a. Layouts are highly useful in JavaFX for organizing elements and objects within a GUI. Numerous layout options are available for easy organization. Layouts are versatile as multiple layouts can be combined (e.g. a FlowPane in a BorderPane).
   b. BorderPane divides an area into five sub-areas: top, bottom, left, right, and center. UI elements can be added to these sub-areas using the setTop(), setBottom(), setLeft(), setRight(), and setCenter() methods.
   c. FlowPane allows for UI elements to be added according to a set boundary where they "flow" automatically into the boundaries. This is useful for situations where elements need to be added to a layout but not necessarily ordered.
   d. GridPane allows for UI elements to be arranged in a grid with set columns and rows. This is helpful in situations such as displaying files.

6.2.2. Implementation Samples

The following blocks of Java code are examples of base JavaFX implementation in a GUI. These show how JavaFX features and classes would be implemented in the application.

1. Application and Scene

```java
public class App extends Application {
	pulbic static void main(String[] args) {
		launch(args);
	}
	@Override
	public void setUp(Stage appStage) {
		appStage.setTitle("SwiftShare");
		Text title = new Text(20, 50, "Welcome to SwiftShare");
		Button sendButton = new Button();
		sendButton.setText("Send");
```

```
                Button receiveButton = new Button();
                receiveButton.setText("Receive");
                Group grp = new Group();
                Scene welcomeScene = new Scene(grp, 600, 500, Color.WHITE);
                grp.getChildren().add(title);
                grp.getChildren().add(sendButton);
                grp.getChildren().add(receiveButton);
                appStage.setScene(welcomeScene);
                appStage.show();
        }
}
```

2. Buttons

Java

```java
public void setUpButtons() {
        Button sendButton = new Button();
        sendButton.setText("Send");
        EventHandler buttonHandler = new EventHandler<ActionEvent>() {
                public void handle(InputEvent event) {
                        //Code for sending file here
                        event.consume();
                }
        };
        sendButton.setOnAction(buttonHandler);
}
```

3. Layouts with CheckBox

Java

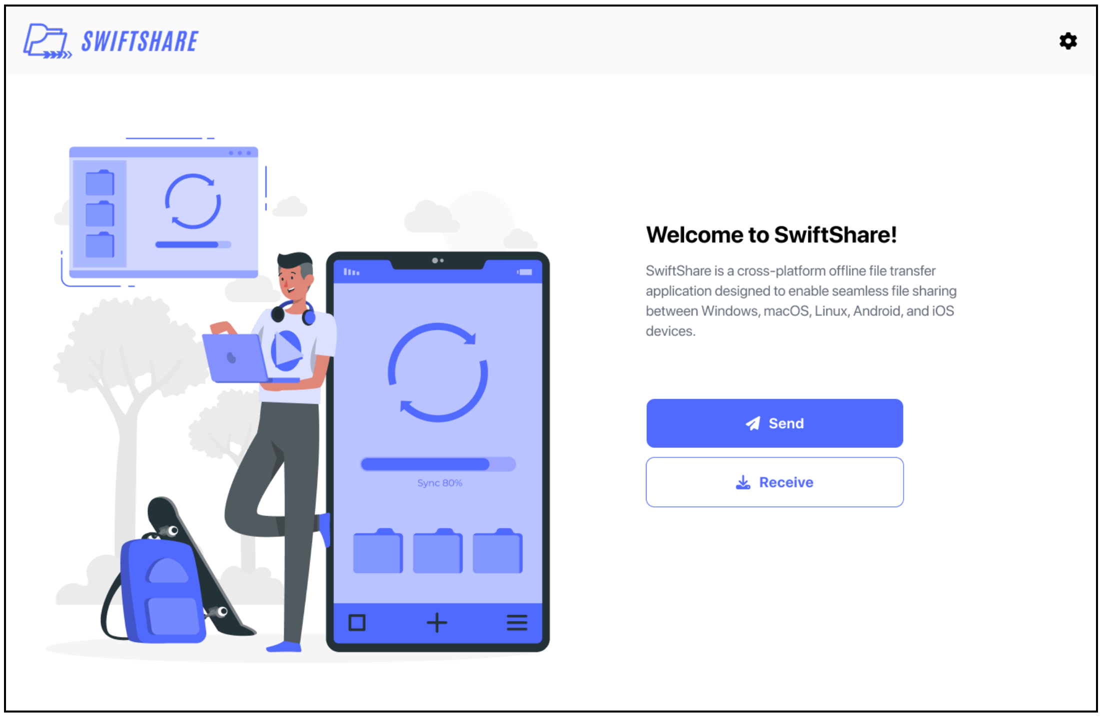
```java
public class FileSelection {
        File selectedFile;
        Icon fileIcon;
        GridPane gridPane;
        CheckBox checkBox;
        public FileSelection(File file, Icon icon) {
                selectedFile = file;
                fileIcon = icon;
                checkBox = new CheckBox(selectedFile.getName());
                gridPane = new GridPane();
                gridPane.add(checkBox, 1, 0)
                gridPane.add(fileIcon, 2, 0);
        }
        public File getFile() {
                return selectedFile;
        }
        public GridPane getPane() {
                return gridPane;
        }
}

public void setLayouts(ArrayList<File> fileList) {
        BorderPane borderPane = new BorderPane();
        FlowPane flowPane = new FlowPane();
        //Set flow pane constraints
        GridPane gridPane = new gridPane();

        borderPane.setTop(contentNode1);
        borderPane.setLeft(new ToolBar());
        borderPane.setRight(contentNode2);
        borderPane.setBottom(new ProgressBar());

        for(File file: fileList) {
                FileSelection selection = new FileSelection(file, <file icon>)
                flowPane.getChildren().add(selection.getPane());
        }

        borderPane.setCenter(flowPane);
}
```

4. Progress Bar

```java
public void setProgressBar() {
        ProgressBar bar = new ProgressBar();
        /*Initialize with 100% Progress*/
        bar.setProgress(1);
}
```

6.3. Screen Images and Action Descriptions

PC Platform

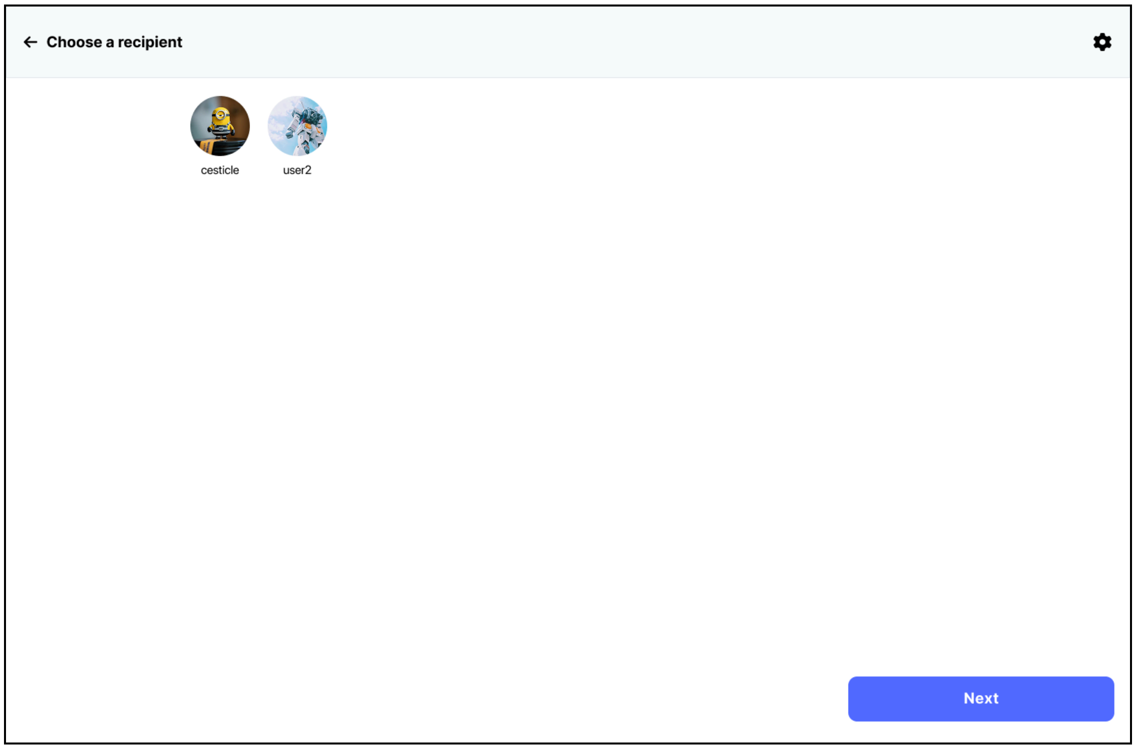


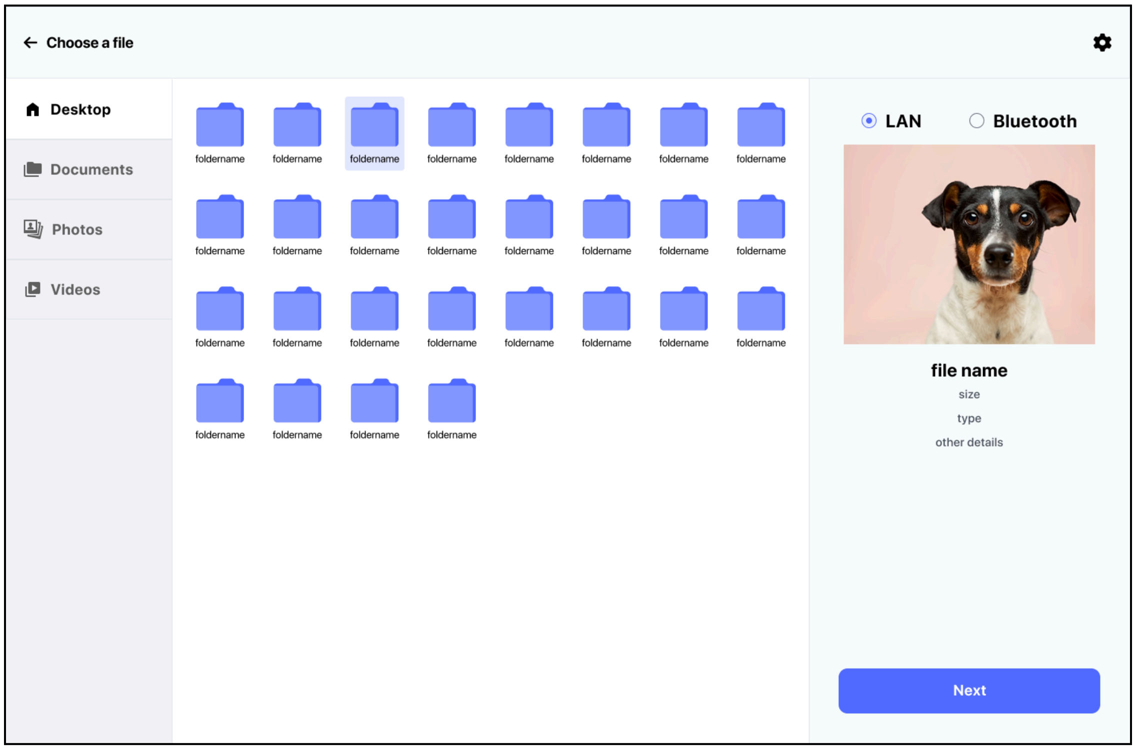
Welcome to SwiftShare!

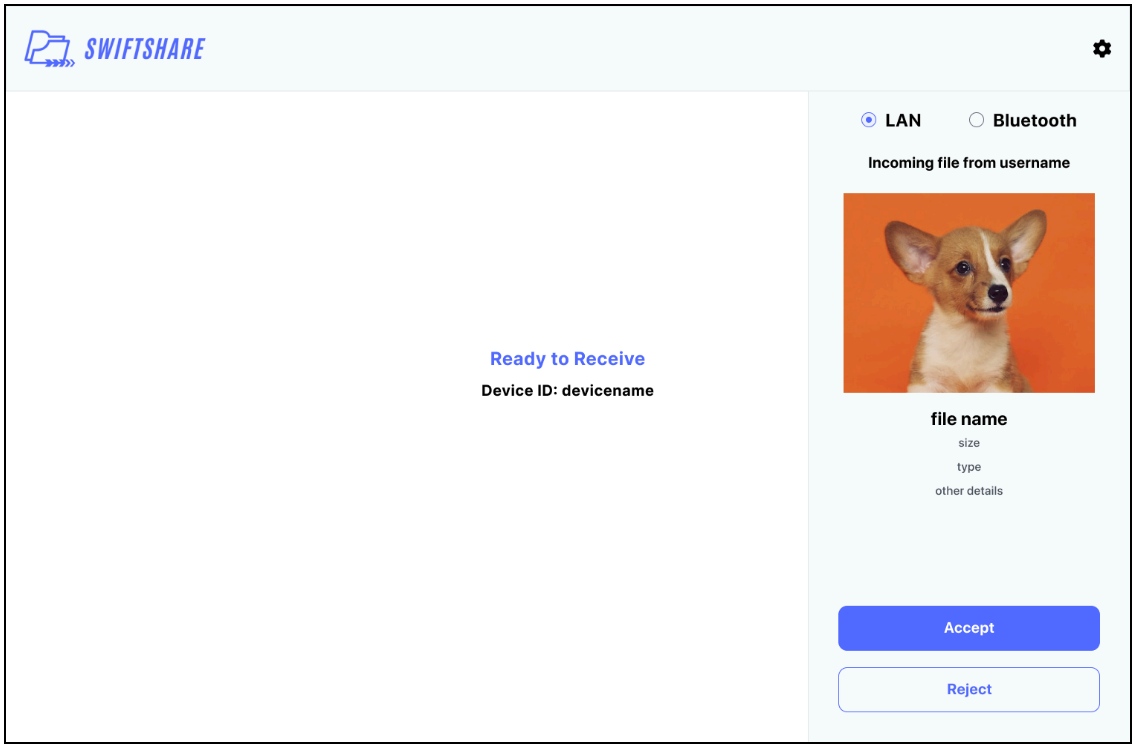
SwiftShare is a cross-platform offline file transfer application designed to enable seamless file sharing between Windows, macOS, Linux, Android, and iOS devices.

Home Page

- The landing page of the application
- Allows users to choose between sending or receiving files
- Shows a list of nearby devices available for file transfer
- Toggle switch for Bluetooth / LAN connections
- Access to Settings for customization

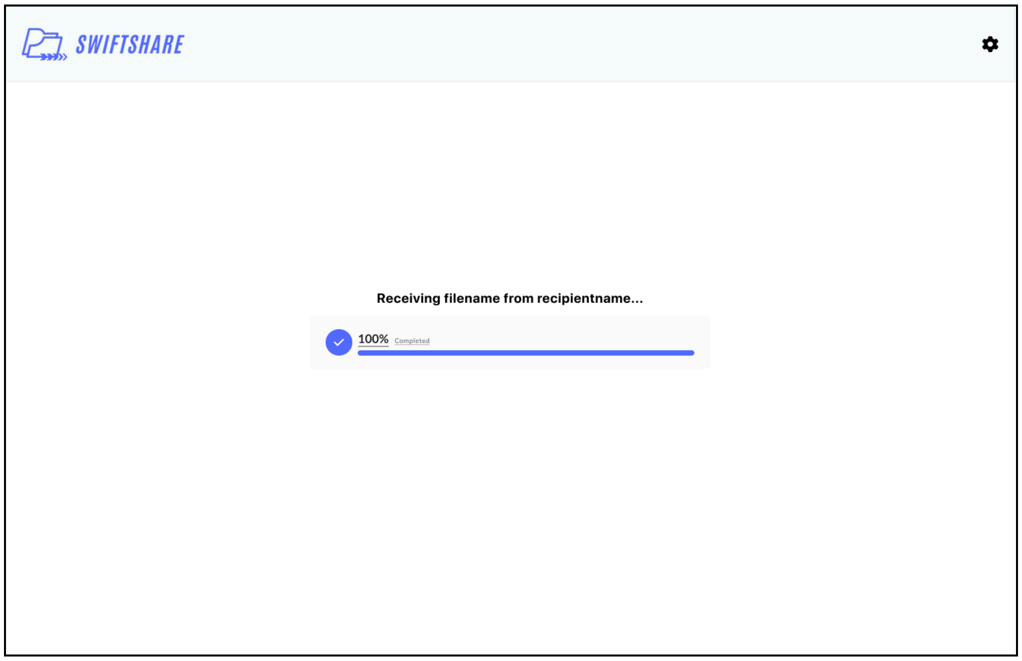

Choose a recipient
cesticle
user2
Next


Choose a file
Desktop
Documents
Photos
Videos
foldername
LAN
Bluetooth
file name
size
type
other details
Next

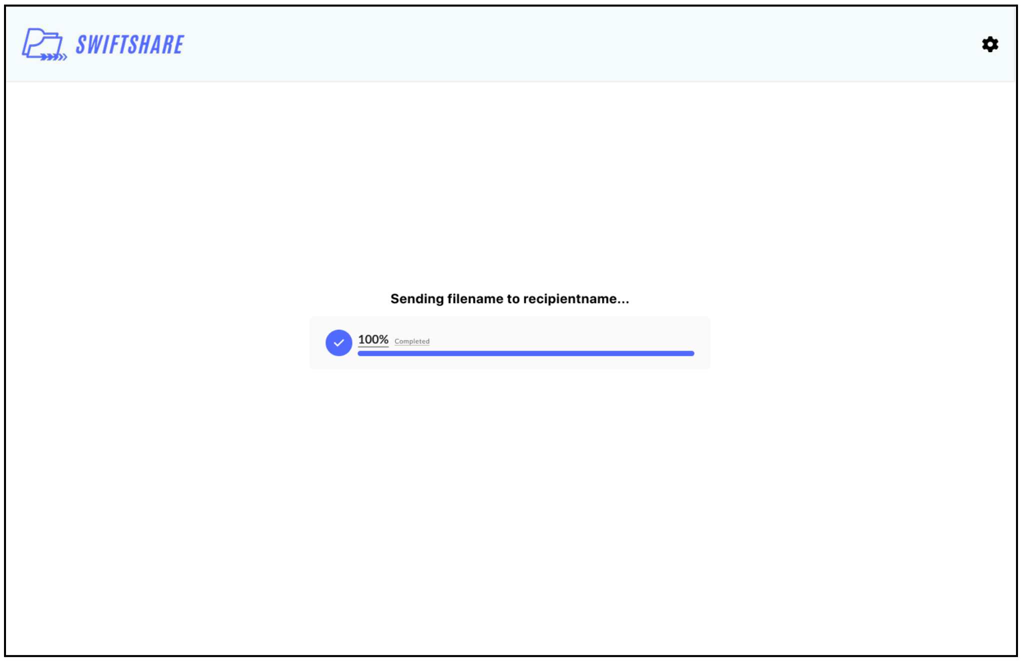


File Selection Screen

- File-explorer like UI which enables users to browse their device for files they want to send
- Includes a drag-and-drop feature for quick file selection
- Categorizes files into images, videos, documents, etc. in order for easy access
- Capability to select multiple files
- Displays the details of selected files (e.g. name, size, format).



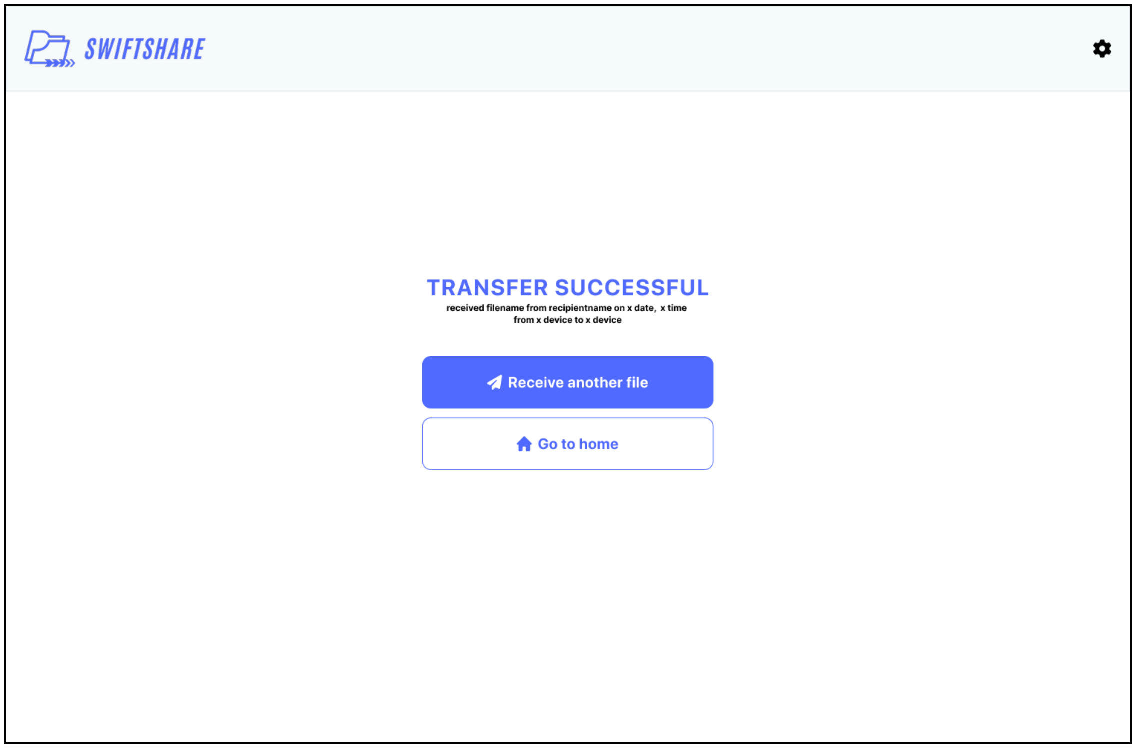
Receiving filename from recipientname...

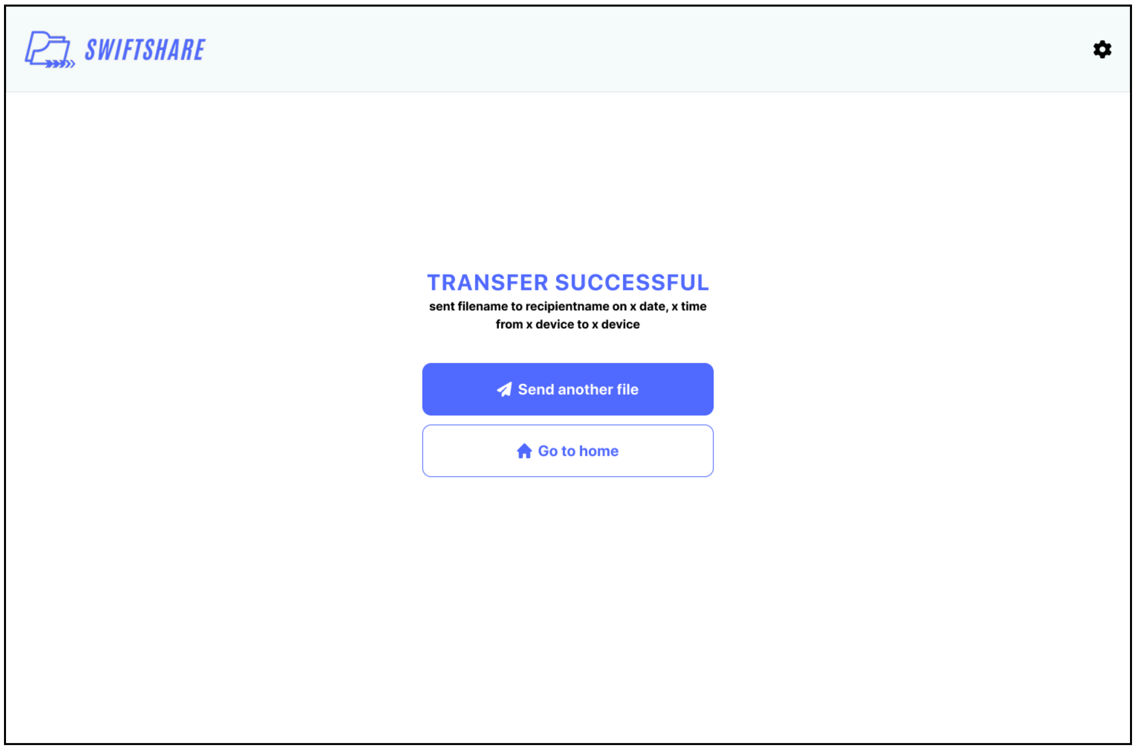


Sending filename to recipientname...

Transfer Screen

- Show ongoing file transfer processes
- Utilizes progress bars as a visual indicator of file transfer completing percentage
- Displays sender/receiver details and an estimated time for transfer completion
- Cancel buttons to allow termination of a file transfer
- Notification upon success or failed transfer


SWIFTSHARE
TRANSFER SUCCESSFUL
received filename from recipientname on x date, x time
from x device to x device
Receive another file
Go to home


SWIFTSHARE
TRANSFER SUCCESSFUL
sent filename to recipientname on x date, x time
from x device to x device
Send another file
Go to home

Transfer History

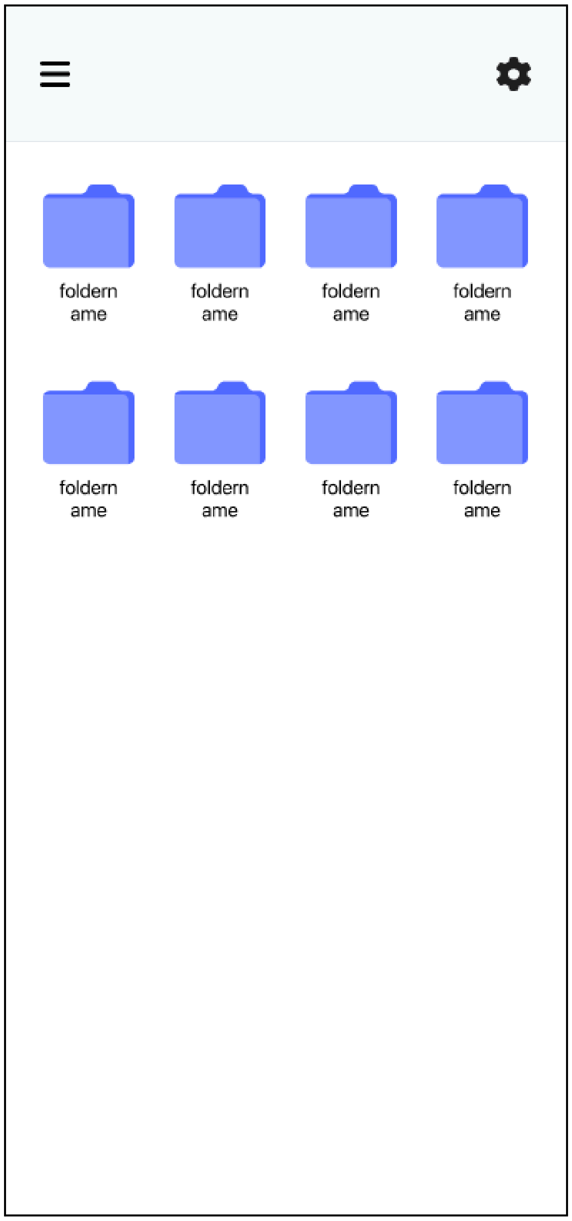
- Displays list of previously transferred files
- Search and filter options to locate past transfers easily

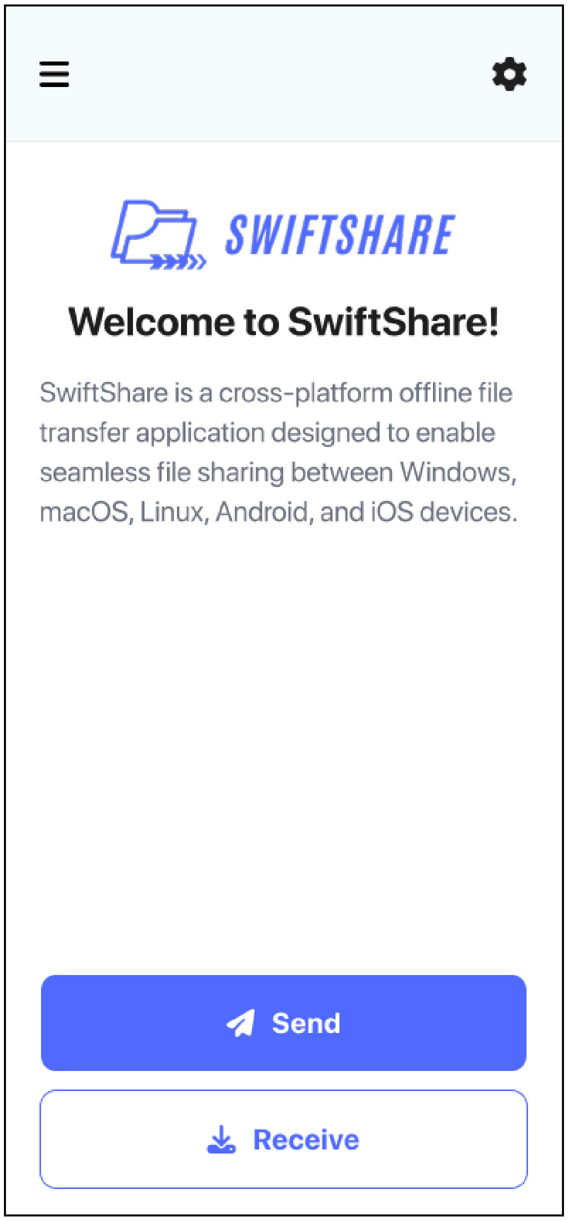
Settings

- Allows users to configure and customize the different parameters of the app
- Change default transfer mode (BT or LAN)
- Dark/light mode toggle for UI
- Adjust visibility and privacy settings for device discovery

As of now, Settings Screen image is being debated as to what are the user-customizable features such as the ones listed above.

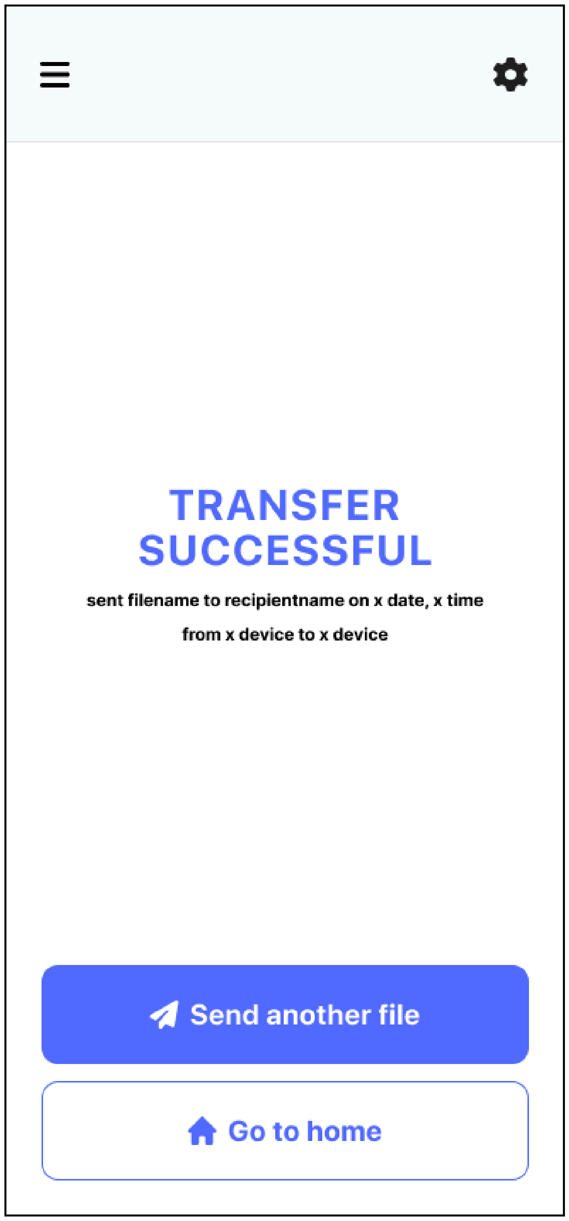
Mobile Platform

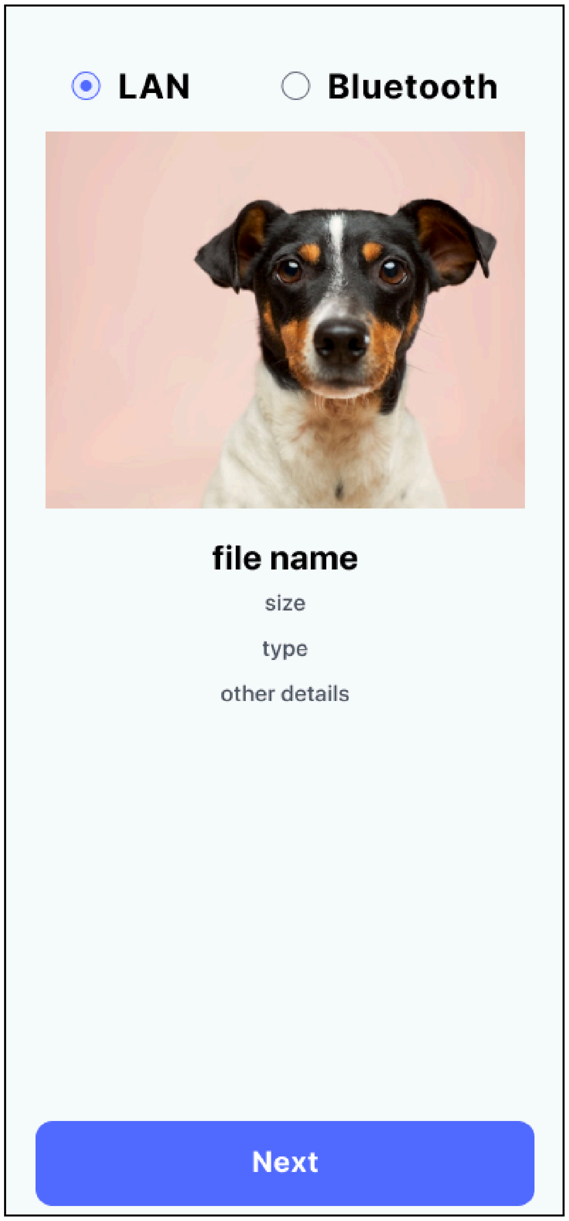


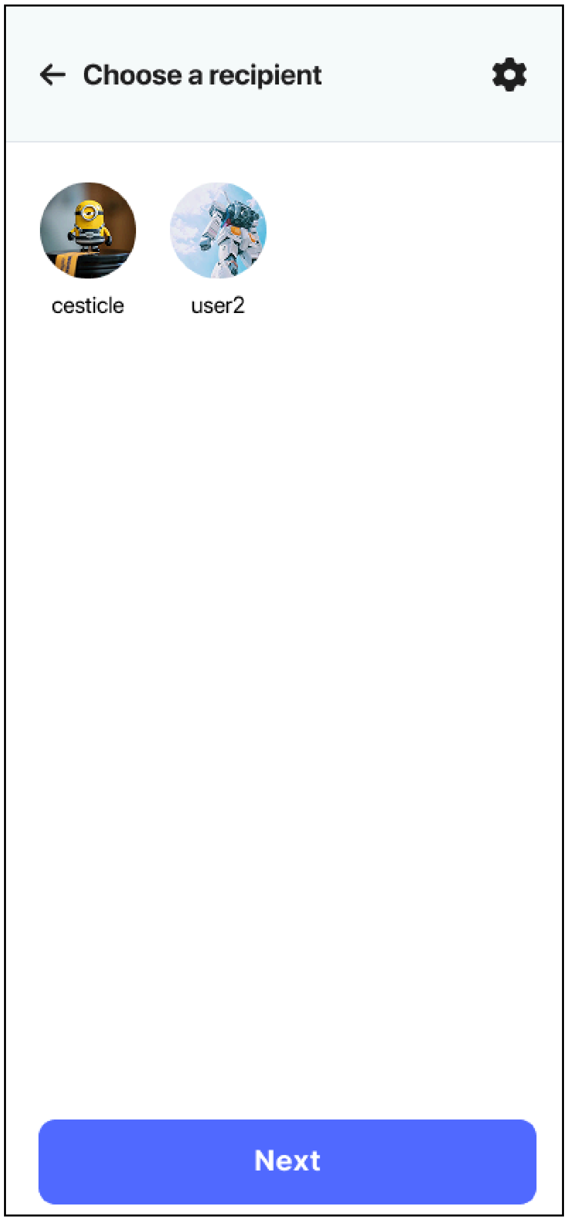


Welcome to SwiftShare!

SwiftShare is a cross-platform offline file transfer application designed to enable seamless file sharing between Windows, macOS, Linux, Android, and iOS devices.


TRANSFER SUCCESSFUL
sent filename to recipientname on x date, x time
from x device to x device
Send another file
Go to home


LAN
Bluetooth
file name
size
type
other details
Next


Choose a recipient

7. REQUIREMENTS MATRIX

Functional Requirements	Components
File Transfer	
REQ-1: Allow users to send files via LAN and Bluetooth	File Transfer Manager, User Interface Component
REQ-2: Ensure data integrity during file transfers. No loss of Data.	File Transfer Manager, Security Component
REQ-3: Support large file transfers with chunking	File Transfer Manager, Data Storage Component.
Device Discovery	
REQ-4: Scan for available devices in range	Device Discovery & Connection Component
Transfer Progress	
REQ-5: Display real-time transfer progress	User Interface Component, File Transfer Manager
REQ-6: Show estimated time for completion and allow cancellation	User Interface Component, File Transfer Manager
File Management	
REQ-7: Allow users to select files for transfer	File Management Component, Data Storage Access Component
REQ-8: Save received files in the appropriate location	File Management Component, Data Storage Access Component
Data Storage & Logging	
REQ-9: Store transfer history and logs	Data Storage Component, File Management Component
REQ-10: Data Storage Component, File Management Component	System Logging & Error Handling Component

8. ITERATIONS

Iteration	1	Title	Project Setup and Core Frameworks
Timeline	Week 1		
Goals	• Set up the **project repository and development environment.** • Establish a basic **JavaFX GUI structure for the desktop app.** • **Configure Maven** for dependency management. • **Set up Android Studio** and SDK for mobile development. • Research and test Bluetooth and LAN transfer protocols.		
Deliverable	• GitHub repository with a structured project setup. • **Placeholder JavaFX UI with basic components.** • Confirmed dependency management using Maven. • **Functional Android project structure with initial UI.** • Initial research findings on LAN and Bluetooth file transfer feasibility.		
Remarks			

Iteration	2	Title	Bluetooth File Transfer Implementation
Timeline	Week 2-3		
Goals	• Implement Bluetooth file transfer using **BlueCove** for desktops. • Develop Android Bluetooth transfer using the **Android Bluetooth API.** • Ensure device discovery and pairing functions work properly. • Test transfer reliability between devices.		
Deliverable	• A working **Bluetooth file transfer between two desktop devices.** • Bluetooth pairing and file transfer between Android devices. • Desktop-to-Android Bluetooth connectivity tests.		
Remarks			

Iteration	3	Title	LAN File Transfer Implementation
Timeline	Week 4-5		
Goals	• Develop a **client-server model for file transfer via LAN.** • Implement file selection and transfer logic using Java NIO. • Ensure cross-platform connectivity between Windows, macOS, and Linux. • Basic Android LAN file transfer logic.		
Deliverable	• **A working file transfer between two desktop devices.** • Basic client-server communication **using Java NIO.** • Android LAN transfer prototype.		
Remarks			

Iteration	4	Title	GUI & User Experience Enhancement
Timeline	Week 6-7		
Goals	● Improve JavaFX UI with: ○ File selection ○ Drag-and-drop support ○ Transfer progress bar ○ Notifications (success/failure) ● Enhance Android UI with Material Design. ● Ensure UI components are wired to backend transfer logic. ● Implement file path caching for frequently transferred files. ● Develop a recent transfers history for quick access.		
Deliverable	● Fully enhanced and functional JavaFX UI with interactive transfer features. ● Improved Android UI with intuitive design. ● File path caching feature. ● Recent transfers history UI on desktop and mobile.		
Remarks			

Iteration	5	Title	Additional Features, Final Testing & Deployment
Timeline	Week 8		
Goals	● *Implement file conversion (e.g. HEIC → JPG, Word → TXT).* ● *Develop a file compression tool (ZIP or custom compression).* ● Conduct extensive testing across platforms. ● **Fix any remaining bugs and performance issues.** ● Package the final build and create a user guide.		
Deliverable	● *Fully functional file conversion and compression features.* ● **Finalized GitHub repository with documentation.** ● User guide and installation instructions. ● Final presentation and project submission.		
Remarks	This part is to be done once all base functionality and UI/UX integration of the application is completed.		

Fun fact. The word client was mentioned over 120 times throughout this document.