Trabajo Final Programación Distribuida y Tiempo Real
Socket Nativos en Android

Alumno: Cazorla, Ignacio Nicolás
Legajo: 13926/1

Contenidos

Introducción

“Un socket es un canal generalizado de comunicación entre procesos. Al igual que una tubería (pipe), un socket se representa como un descriptor de archivo. A diferencia de las tuberías, los sockets permiten la comunicación entre procesos no relacionados, e incluso entre procesos que se ejecutan en diferentes máquinas y se comunican a través de una red. Los sockets son el principal medio de comunicación con otras máquinas; programas como **telnet**, **rlogin**, **ftp**, **talk** y otros programas familiares de red utilizan sockets.

No todos los sistemas operativos admiten sockets. En la Biblioteca C de GNU, el archivo de encabezado sys/socket.h existe independientemente del sistema operativo, y las funciones de socket siempre existen, pero si el sistema no admite realmente sockets, estas funciones siempre fallarán.” (“GNU C Library - Capítulo 16”, n.d., párr. 1-2).

Android es un sistema operativo basado en GNU/Linux. Cómo se indica en el párrafo anterior, este último utiliza la librería **glibc** para implementar muchas de sus funcionalidades. De esta manera, muchas de las funciones que provee esta librería deberían encontrarse implementadas también en Android.

El presente documento pretende exhibir una experimentación con sockets nativos en el sistema operativo Android. Se va a indicar cómo se crea y configura un proyecto de estas características, herramientas y/o librerías para llevar a cabo el desarrollo y finalmente la implementación de un cliente y un servidor que se comunican mediante sockets, con código C originalmente utilizado en sistemas GNU/Linux.

Puesta a punto del entorno de desarrollo

En este apartado se mencionan las herramientas necesarias para poder llevar adelante este proyecto.

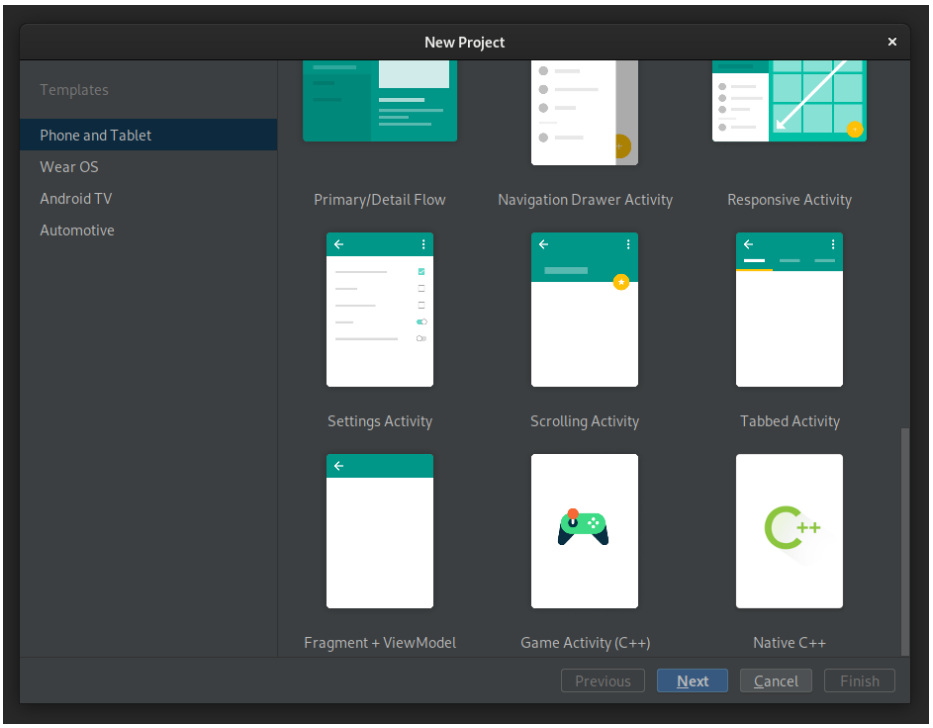
La herramienta básica es **Android Studio**[1]. Se puede obtener desde su sitio web. Contiene todo lo necesario para el desarrollo (IDE, ADB, emulador, etc), y este material está confeccionado en base a ella, por lo que se recomienda su uso. Sin embargo, es posible realizar la instalación de las herramientas necesarias por separado.

[1] La versión de Android Studio que se va a utilizar es **Android Studio Dolphin | 2021.3.1 Patch 1**, Build #AI-213.7172.25.2113.9123335, built on September 30, 2022, para un sistema operativo GNU/Linux.

Luego, **las herramientas más importantes** son **NDK** (*Native Development Kit*) y **Cmake**. NDK permitirá hacer uso de la biblioteca nativa *Bionic* (adaptación de glibc). Cmake, por su parte, sirve para el proceso de compilación. Ambas pueden configurarse a partir de las guías del sitio de desarrolladores de Android.

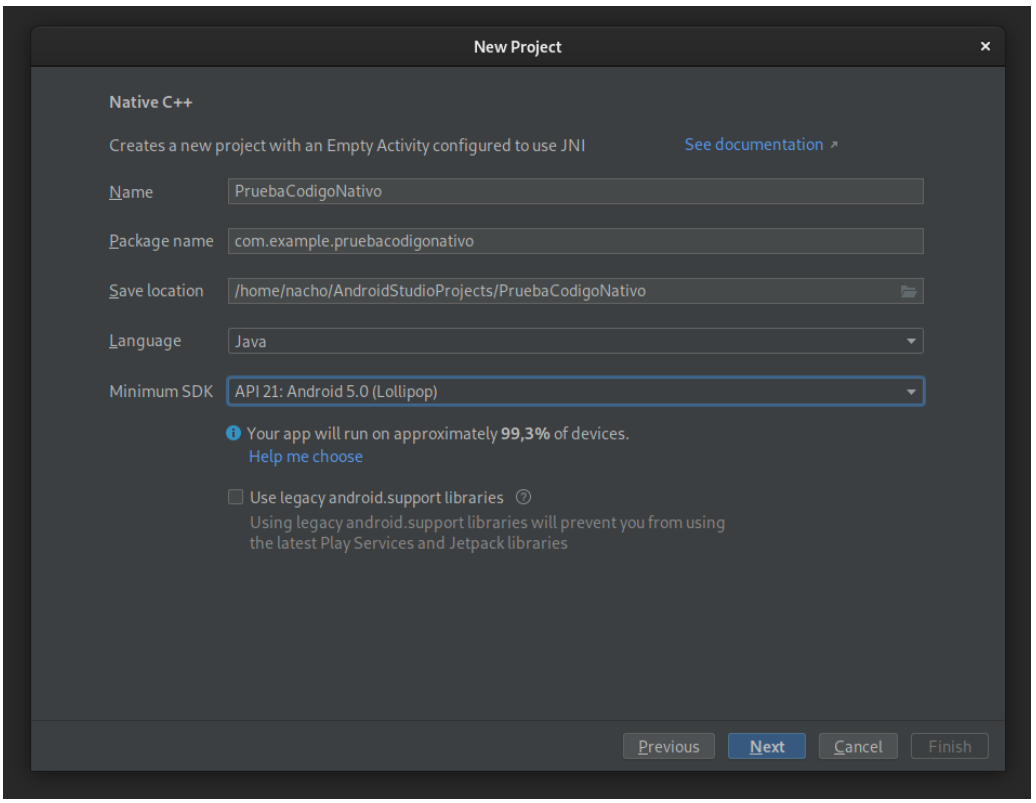
Generar un proyecto Android

Desde **Android Studio** vamos a generar un nuevo proyecto.



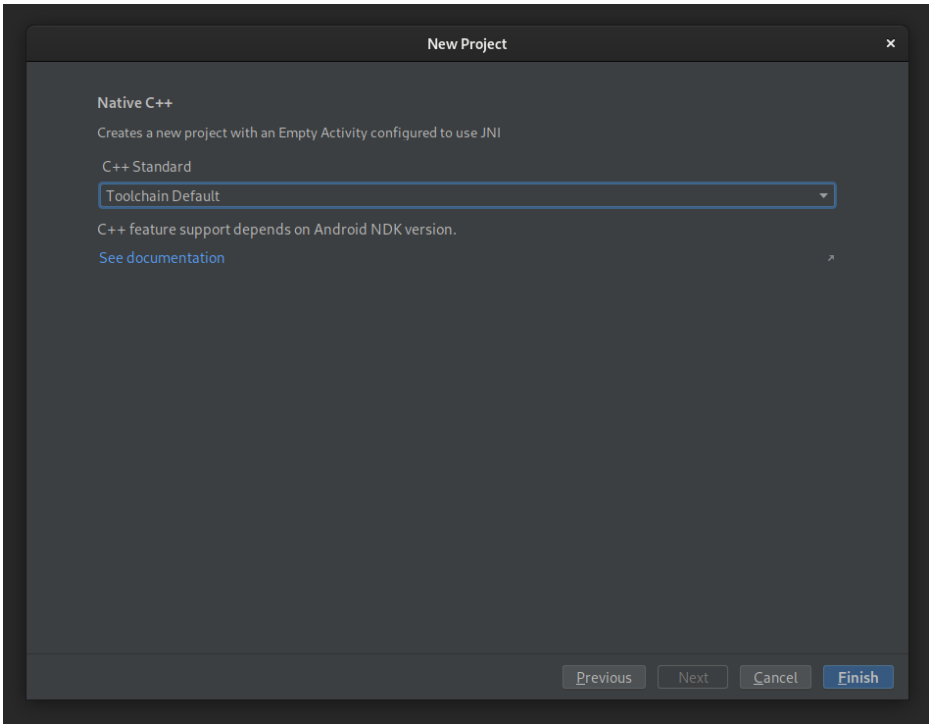
Nuevo proyecto Android, selección de plantilla

Podemos seleccionar la plantilla ***Native* C++** que ya viene configurada para usar JNI (JNI actúa como intermediaria entre el código nativo y el de *bytes* generado Android) o elegir arrancar con una ***Empty Activity***. Para este ejemplo se selecciona la plantilla ***Native* C++**.



Nuevo proyecto Android, configuración básica

Indicamos el nombre del proyecto, el lenguaje (Java o Kotlin) y el SDK mínimo.



Nuevo proyecto Android, selección de la versión de biblioteca de C++

Podemos seleccionar “Finish”, sin hacer cambios. A continuación se generará el proyecto.

Java Native Interface (JNI)

"JNI es la interfaz nativa de Java, que define una forma para que el código de *bytes* que Android compila a partir de código administrado (escrito en los lenguajes de programación Java o Kotlin) interactúe con el código nativo (escrito en C/C++)" ("JNI: Interfaz Nativa de Java", 2021).

En otras palabras, "La JNI es una interfaz de programación nativa que permite que el código Java que se ejecuta dentro de una Máquina Virtual de Java interactúe con aplicaciones y bibliotecas escritas en otros lenguajes de programación, como C, C++ y ensamblador" ("Interfaz Nativa de Java (JNI)", n.d., párr. 1).

Dentro de los escenarios en los que se podría desear utilizar esta interfaz, son destacables la necesidad de obtener mayor performance o si se quiere reutilizar código escrito en los lenguajes previamente mencionados.

En el desarrollo que se va a proponer, JNI permitirá que nuestra aplicación Android (implementada en Java) interactúe con el código nativo. A continuación se presenta un ejemplo básico para comenzar a familiarizarse con esta interfaz. Se puede acceder al código fuente desde la carpeta **“PruebaCodigoNativo”**. Supongamos que se quiere imprimir un *string* que nos provee un código nativo.

Comenzamos en la clase ***MainActivity*** dónde llamamos a ***System.loadLibrary()***, la función que necesitamos para poder cargar el código nativo. Esta función recibe como argumento el nombre de la librería nativa que se desea cargar.

```java
package com.ejemplo.pruebacodigonativo

public class MainActivity extends AppCompatActivity {

    // Used to load the 'prueba_codigo_nativo' library on
application startup.
    static {
        System.loadLibrary("prueba_codigo_nativo");
    }

    // Native method call
    native String saludar();

    // MainActivity methods
    ...
}
```

Clase *MainActivity*.java

Notar que se utiliza un **inicializador de clase estático** de Java (“static{...}”). Esto es un bloque de código que se ejecuta una sola vez, apenas la clase es cargada en memoria y antes de que se generen instancias de la misma. Como su nombre lo indica, el código realizará tareas de inicialización específicas de la clase. Es importante destacar que de esta manera estamos cargando la librería en **tiempo de ejecución**, es decir, de **manera dinámica**. Es posible también realizar la carga de manera estática cargando las librerías en tiempo de compilación, ya que existen sistemas operativos que no permiten la carga dinámica de estas, u otras situaciones donde puede ser deseable realizar la carga de manera estática. JNI cuenta con ambas alternativas. Para los sistemas operativos que no soportan carga dinámica de librerías, ***System.loadLibrary()*** terminará sin cargar la librería indicada.

Siguiendo con el proceso para lograr imprimir desde código nativo, las librerías se definen en el directorio **cpp** en el archivo ***CMakeLists.txt***. Para ello usamos ***add_library()***, indicando el nombre de la librería, si es estática o dinámica y la ruta relativa (a ***CMakeLists.txt***) de los archivos fuente.

```
# For more information about using CMake with Android Studio, read
the documentation:
https://d.android.com/studio/projects/add-native-code.html

# Sets the minimum version of CMake required to build the native
library.
cmake_minimum_required(VERSION 3.18.1)

# Declares and names the project.
project("pruebacodigonativo")

# Creates and names a library, sets it as either STATIC
# or SHARED, and provides the relative paths to its source code.
# You can define multiple libraries, and CMake builds them for
you.
# Gradle automatically packages shared libraries with your APK.

add_library( # Sets the name of the library.
        prueba_codigo_nativo

        # Sets the library as a shared library.
        SHARED

        # Provides a relative path to your source file(s).
        saludo.c)
```

Archivo ./app/src/main/cpp/CMakeLists.txt

El nombre con el que se define la librería en este caso es "**prueba_codigo_nativo**" y es el que se especifica en ***MainActivity***.

Resta por ver el contenido del archivo "**saludo.c**". Este define el código que imprime un *string*, en la función ***Java_com_example_pruebacodigonativo_MainActivity_saludar()***.

```c
#include <jni.h>

JNIEXPORT jstring JNICALL
Java_com_example_pruebacodigonativo_MainActivity_saludar(JNIE
nv *env, jobject thiz) {
    char * hello = "Hola desde C!";
    return (*env)->NewStringUTF(env, hello);
}
```

Código C nativo ./app/src/main/cpp/saludo.c

El nombre del método definido es concatenado con los siguientes elementos:

- el prefijo “Java_”
- el nombre completo de la clase y el paquete
- un “_” separador
- el nombre definido para el método por el desarrollador (también llamado nombre **corto**, en este caso “saludar”)

La implementación JNI de cada Máquina Virtual de Java se encargará de resolver la carga y la resolución del nombre de cada método internamente.

```java
protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);

       binding =
ActivityMainBinding.inflate(getLayoutInflater());
        setContentView(binding.getRoot());
        // Ejemplo de llamada a método nativo
        TextView tv = binding.sampleText;
        tv.setText(saludar());
    }

    //Interfaz para invocar al método nativo
    public native String saludar();
}
```

Clase MainActivity.java

Al invocar el método en la línea “***native String* saludar();**” en ***MainActivity*** la Máquina Virtual de Java busca una función dentro de la librería nativa que coincida con el nombre

corto (el que define el desarrollador) indicado en la invocación. Si el método fuese "sobrecargado" se debería especificar el nombre **completo**.

El resultado de la ejecución de ésta aplicación mostrará como salida el texto "Hola desde C!".

Resultado de la ejecución

Glibc en Android: Bionic

Bionic es una adaptación de **glibc** de GNU/Linux para Android, hecha por Google. Básicamente contiene la librerías de C, de matemática y el *dynamic linker*. *Bionic* es de suma importancia para este proyecto. El hecho de ser una adaptación permite que sea más factible poder utilizar **sockets** de la misma manera que en un sistema operativo GNU/Linux.

Esta adaptación surge de la necesidad de obtener una librería más pequeña para dispositivos (en principio) con menores capacidades, es decir, menor cantidad de memoria y menor poder de procesamiento.

Es posible acceder al "código fuente" y cuenta con una licencia **BSD**.

Desarrollo con sockets nativos en Android

A continuación se describe en tres versiones, una aplicación que toma como base un cliente y un servidor desarrollados en el lenguaje C que se comunican mediante **sockets**. El código de ambos programas se encuentra en la carpeta **"CodigoBaseSocketsC"**. La intención fue replicar la interacción que realizan dichos programas en una aplicación de Android. Se pensó y diseñó este proyecto con la idea de poder trabajar en gran medida con el emulador que provee **Android Studio**.

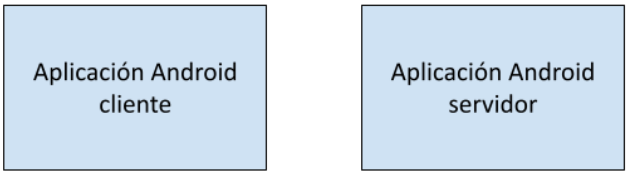
Primera versión

Esta versión se diseñó con la finalidad de analizar la posibilidad y dificultad de utilizar sockets nativos para comunicar procesos en Android. Se utilizaron threads que se comunican en el mismo dispositivo. Tanto cliente como servidor se ejecutan en dos hilos. La interacción y los resultados son sólo visibles desde una pestaña de salida (pestaña "*Run*") de Android Studio.

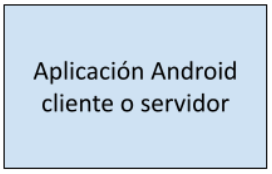
Organización de los procesos y subprocesos

El objetivo principal es poder comunicar dos procesos vía sockets nativos. Entre las alternativas analizadas para organizar tales procesos, se tuvieron en cuenta:

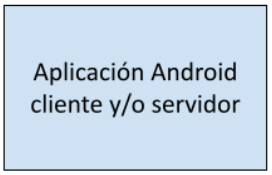
1. Crear dos aplicaciones distintas, es decir, una para el cliente y otra para el servidor.
2. Crear una única aplicación que tenga el código del cliente y el del servidor pero solo pueda ejecutar uno por vez (se deben ejecutar dos instancias del mismo programa, una para ejecutar el servidor y otra para el cliente).
3. Una única aplicación como en la alternativa anterior, pero que permita que tanto cliente como servidor puedan correr a la vez (la misma aplicación, posibilidad de ejecutar a ambos en la misma instancia de la aplicación).



1. Una aplicación para el cliente y otra para el servidor



2. Aplicación que puede ser cliente o servidor (una a la vez).



3. Aplicación que puede ser cliente y/o servidor (puede ser más de una a la vez)

Esquema de implementación

La primera alternativa se descartó inmediatamente. La principal desventaja que representaba, era la poca flexibilidad que tendría cada aplicación por tener solo una funcionalidad (cliente o servidor).
El problema que presentó la segunda, fue que para probarlo únicamente con el emulador se necesitaba dejar a alguna instancia de la aplicación en segundo plano. Por eso, se optó por la tercera alternativa y que así puedan ejecutarse en la misma instancia de una aplicación a la vez.

En Android cada aplicación se ejecuta en un único proceso, con un único hilo de ejecución. Este hilo "principal" se encarga de todas las tareas relacionadas a la interfaz de usuario. Es importante (y una buena práctica) delegar las nuevas tareas creando hilos para evitar saturar al hilo principal. Invocar operaciones que requieran un tiempo prolongado para completarse en el hilo principal puede llevar a que se bloquee la interfaz de usuario. Por este motivo, tanto proceso cliente como servidor se ejecutarán en un hilo cada uno.

Manejo de subprocesos (hilos)

Las alternativas mencionadas en este apartado surgieron de la necesidad de poder separar la ejecución tanto del cliente como del servidor del hilo principal.

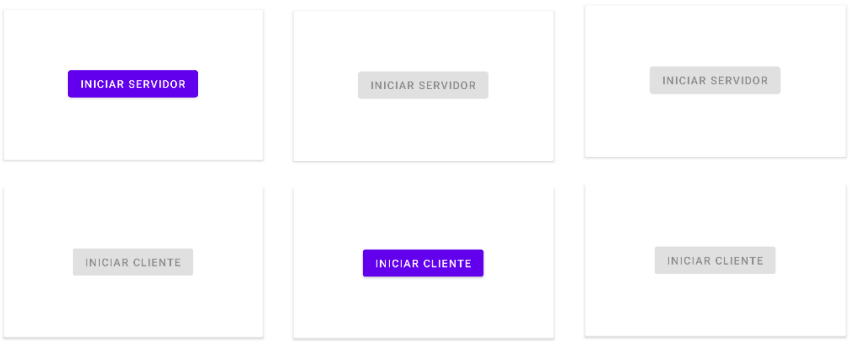
La manera adecuada de manejar hilos en Android actualmente, es utilizando las interfaces ***Executor*** y ***ExecutorService***.
Executor es una interfaz básica que tiene un método que acepta una tarea (implementada como un objeto ***Runnable***) y la ejecuta de manera asíncrona en algún hilo disponible. En resumen, sirve para abstraer el manejo de subprocesos sin preocuparse por el resultado o estado de las tareas. ***ExecutorService*** por su parte, extiende la interfaz ***Executor*** y agrega métodos para manejar la terminación y los estados de las tareas asincrónicas. Esta interfaz resulta más útil cuando se necesitan manejar varias tareas y tener control sobre sus estados y resultados.

Vale la pena mencionar que también existe la clase ***Service***. Según el sitio de desarrolladores de Android: "Un *Service* es un componente de una aplicación que puede realizar operaciones de larga ejecución en segundo plano y que no proporciona una interfaz de usuario." "**Diferencia entre un servicio y un subproceso**: Un servicio es simplemente un componente que puede ejecutarse en segundo plano, incluso cuando el usuario no está interactuando con tu aplicación. Por eso, solo debes crear un servicio si es lo que necesitas. En cambio, es necesario crear un subproceso si debes trabajar fuera de tu subproceso principal solamente mientras el usuario interactúa con tu aplicación."

En conclusión, ***Service*** es un componente más de Android que puede ejecutarse en segundo plano, pero no ejecuta en un nuevo hilo.

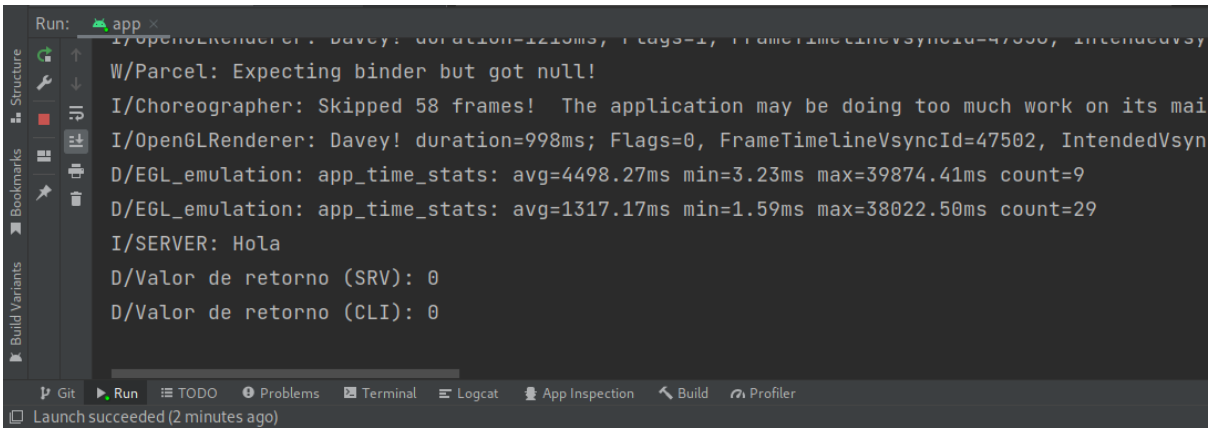
Implementación

El código para esta versión inicia desde **MainActivity**. Allí se definen *listeners* para dos botones. Un botón permite iniciar un servidor que escucha en todas las interfaces disponibles y el puerto 30000. El otro botón, ejecuta al cliente, que se conecta a dicho puerto y a la dirección IP de ***loopback*** para enviar un *string* ("Hola"). El servidor debe recibir este mensaje y mostrarlo en la consola de ejecución de Android Studio (pestaña "Run").



1. Iniciar servidor **2. Iniciar cliente** **3. Fin de la interacción**

Como resultado se puede ver en la salida el mensaje enviado:



Mensaje recibido por el servidor en consola

La línea "I/SERVER: Hola" refleja el contenido del *buffer* luego de recibir un mensaje del cliente en el servidor.

El código tomado como base tuvo que sufrir muy pocas modificaciones para poder funcionar correctamente. Entre los cambios realizados es posible mencionar que se quitaron llamadas a funciones (*printf()* y *exit()*) y se reemplazaron por otras. Por ejemplo, para reemplazar a la función *exit* se agregaron retornos desde la función nativa que actúa como cliente/servidor. Los valores de retorno representan el estado de la comunicación (0 Correcto, otro caso indica error). Respecto de la impresión en consola (*printf()*) se utilizaron los "*logs* de Android".

Vale la pena mencionar que el servidor está escuchando en todas las interfaces de red, la dirección que se especifica es **INADDR_ANY**[2] (0.0.0.0). Así, se puede establecer una conexión por cualquiera de las interfaces con IPv4 del dispositivo. Por el momento, sólo se está usando la interfaz de *loopback* para establecer comunicación.

[2] Es posible profundizar más en el funcionamiento de INADDR_ANY en "Linux IPv4 protocol implementation".

Resultados

El objetivo de esta primera versión fue probar el funcionamiento de código nativo para utilizar sockets en Android. Dicho objetivo fue alcanzado.

Segunda versión

El desarrollo continúa con el diseño e implementación de una interfaz de usuario. La intención es que sea posible interactuar con la aplicación y visualizar los resultados dentro de la misma. También se manejan los datos de retorno tanto del cliente como del servidor.

Configurar el *pool* de hilos

Crear hilos es costoso, entonces se debe evitar crearlos y destruirlos continuamente. Para ello inicializamos un *pool* de hilos en una nueva clase ***MyApplication*** que extiende de ***Application***. Estos hilos pasan a ser "globales" y se crean al iniciar la aplicación. La línea ***Executors.newFixedThreadPool(2)*** indica que se deben crear dos nuevos hilos.

```
public class MyApplication extends Application {
// Crea dos hilos globales para luego ejecutar una instancia
que actua como cliente y otra de servidor
       public ExecutorService executorService =
Executors.newFixedThreadPool(2);
}
```

Creación de hilos "globales"

Luego, es necesario indicar que ***MyApplication*** reemplaza a ***Application*** en el archivo **AndroidManifest.xml**.

```
<application
   android:name=".MyApplication"
   ...
```

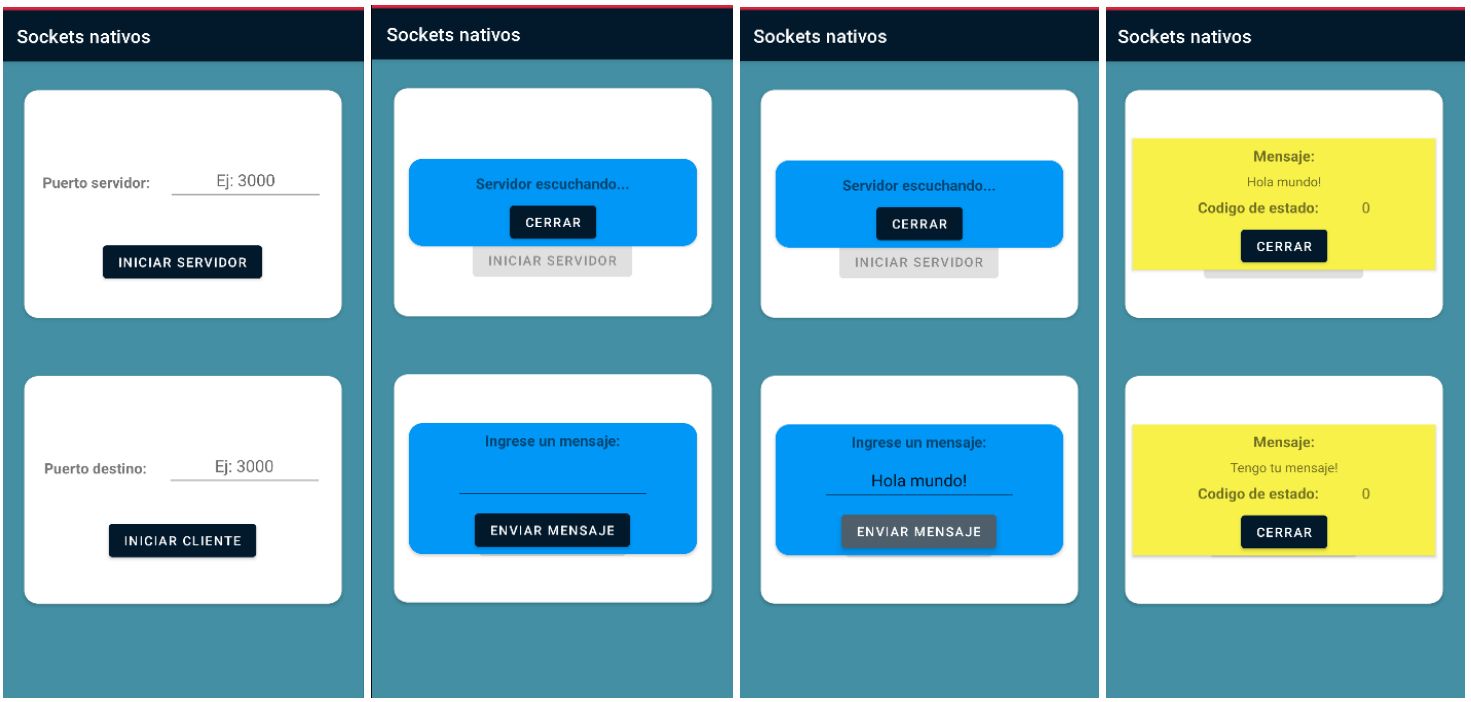
Indicar la clase que reemplaza a *Application*

Manejo de respuestas de cliente y servidor

Los resultados que se obtienen de cada proceso (servidor o cliente) no son visibles fuera de la ejecución de cada hilo, por lo que deben ser manejados con un *callback* y ***ViewModels*** (ver "Uso de *viewmodels*").

Implementación

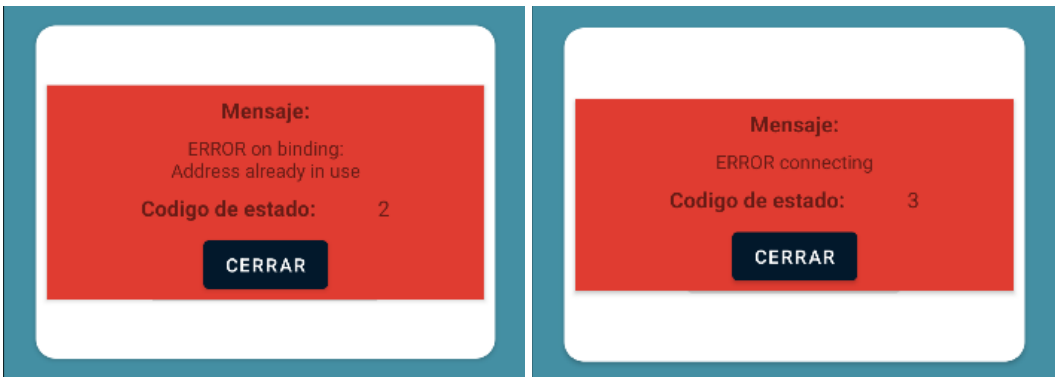
La implementación en este apartado corresponde en mayor medida a la interacción con la aplicación por parte del usuario.



Ejemplo: envío de mensaje desde cliente

Resultados

Se logra una interfaz que permite definir puertos para el cliente y el servidor. También se puede ingresar el mensaje para enviar desde el cliente y visualizar el mensaje recibido desde el otro proceso y/o errores durante la comunicación.
Cabe destacar que con la solución propuesta, luego de establecer una comunicación por un puerto, no es posible reutilizarlo inmediatamente (ver "*Address already in use*").



Mensajes de error

Tercera versión

Esta versión agrega la posibilidad de establecer la conexión entre procesos dentro de una LAN.

Conexión entre dispositivos

Las alternativas utilizadas para probar la conexión entre lo dispositivos fueron:

1. Emulador + Utilidades del sistema host
2. Emulador + Dispositivo Android
3. Dispositivo Android + Dispositivo Android

Breve y progresivamente se describirá cada alternativa para lograr la conexión dentro de una LAN. La más sencilla de probar es la número 3, pero requiere de dos dispositivos móviles.

Si se decide utilizar el emulador para la conexión, debido a que este se encuentra en una LAN virtual, deben efectuarse pasos previos. Se necesita definir una redirección para que el emulador pueda comunicarse fuera de esa red virtual. Antes de detallar las alternativas número 1 y 2, debemos conectarnos[3] al emulador de Android desde una terminal.

```
$ telnet localhost 5554
```

Una vez conectado, el usuario debe autenticarse. La forma de hacerlo es obtener el token que se encuentra en **'/home/user/.emulator_console_auth_token'** y luego ejecutar el comando **auth** brindándole dicho token.

[3] Más detalles sobre la conexión con el emulador en "Cómo enviar comandos de la consola del emulador".

```
auth 123456789ABCdefZ
```

Cada vez que la consola muestre **OK**, estará lista para aceptar más comandos.

Una vez dentro del emulador debemos especificar la redirección. El comando para hacerlo es **redir add <*protocol*>:<*host-port*>:<*guest-port*>** donde:

- <*protocol*>: protocolo de capa de transporte (tcp |udp)
- <*host-port*>: indica el puerto de la máquina host
- <*guest-port*>: indica el puerto del emulador

Es importante tener en cuenta que como la conexión fue diseñada para un solo intercambio de mensajes, luego de utilizar un puerto no es posible reutilizarlo inmediatamente. Es necesario agregar una redirección cada vez que queramos establecer una nueva conexión, esperar a que el socket esté disponible nuevamente, o reiniciar la ejecución de la aplicación (ver "*Address already in use*").

1. Emulador + Utilidades del sistema host

Una vez realizados los pasos anteriores es posible probar la conexión desde una terminal del *host*. Se utilizará el comando **nc**.

Paso 1: Se inicia el servidor en el puerto 3000 en la IP de la LAN virtual del dispositivo.

Iniciar servidor (puerto 3000, IP de LAN del simulador)

Paso 2: Se define la redirección de la siguiente manera entre el puerto 4000 y el puerto en el que va a escuchar el servidor, en este caso el 3000.

```
redir add tcp:4000:3000
OK
```

Definición de una redirección en el emulador

Por último, se debe ejecutar el comando **nc** e ingresar un mensaje, en este caso "Hola desde la terminal del *host*!". La respuesta del servidor debería ser "Tengo tu mensaje!".

```
$ nc localhost 4000
Hola desde la terminal del host!
Tengo tu mensaje!
```

Ejecución del comando en el *host*

En la aplicación, el servidor debería mostrar el mensaje que se envió desde la terminal del *host*.

Salida del servidor (comunicación mediante nc)

2. Emulador + Dispositivo Android

De manera similar a la alternativa 1, es posible lograr comunicar un dispositivo Android con el emulador. Utilizaremos como intermediario al *host*. Para lograr comunicar ambos dispositivos utilizaremos nuevamente el comando **nc**. Además haremos uso del comando **mkfifo**. Este comando nos permite crear un *pipe* nombrado, un tipo de archivo especial, que permite que procesos independientes se comuniquen.

Partiendo de los pasos 1 y 2 del inciso anterior, para inicializar al servidor y para definir la redirección, a continuación debemos ejecutar los siguientes comandos en la máquina *host* del emulador:

```
$ mkfifo out
$ nc -l -s 192.168.0.174 -p 3000 <out | nc localhost 4000 > out
```

`nc -l -s ip-host -p 3000` : inicia un servidor ("servidor del *host*") que escucha en el puerto 3000 en el *host*, dónde ip-host tiene que ser una dirección IP de LAN del mismo.
`nc localhost 4000` : actúa como "cliente del emulador" desde el *host* que emula hacia el dispositivo Android emulado.
El *pipe* anónimo (|) : comunica la salida del "servidor del *host*" hacia el "cliente del emulador".
El "cliente del emulador" luego redirige la respuesta que recibe del servidor de la aplicación hacia el *pipe* nombrado (*out*) y finalmente el contenido de *out* alimenta con su redirección al "servidor del *host*". De esta manera, el cliente original (dispositivo Android) recibe la respuesta desde el emulador.

3. Dispositivo Android + Dispositivo Android

Esta alternativa es la más sencilla. No requiere ninguna herramienta intermediaria como en los casos anteriores.

Implementación

Para poder lograr la comunicación entre dispositivos Android dentro de una LAN, sólo fue necesario permitir especificar qué direcciones IP usar.

Desde la perspectiva del servidor contamos con un *spinner* que permite seleccionar una dirección IP en la que el servidor escucha por conexiones. Además, se puede especificar en qué puerto queremos que escuche nuestro servidor. Desde la perspectiva del cliente, podemos indicar una dirección IP a la cuál queremos conectarnos (la del servidor) y el puerto. Luego, podemos enviar un mensaje desde el cliente hacia el servidor. Lo que ocurre una vez que indicamos la interfaz de red que queremos utilizar en el servidor, es que se le indica dicha interfaz al código nativo. La opción "Todas las interfaces" especificará como interfaz **INADDR_ANY** (0.0.0.0), cómo se indica sobre el final de la primera versión. Nos sirve para recibir por cualquier interfaz que el dispositivo tenga configurada (para IPv4).

Resultados

Se logra establecer una comunicación mediante red de área local entre dispositivos Android.

Sección adicional

Permitir conexión a internet

Para poder establecer conexiones desde la aplicación, es necesario agregar permisos de internet en el archivo ***AndroidManifest.xml***. Deben agregarse las líneas
<uses-permission android:name="android.permission.INTERNET" /> y
<uses-permission android:name="android.permission.ACCESS_NETWORK_STATE" />

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

   <uses-permission
android:name="android.permission.INTERNET" />
   <uses-permission
android:name="android.permission.ACCESS_NETWORK_STATE" />
...
>
```

Modificación de AndroidManifest.xml

Logs desde código nativo para *debugging*

Para poder imprimir en pantalla desde el código nativo es necesario utilizar la librería ***log.h*** de Android. La función que se debe invocar es **__android_log_print(priority, tag, format_string, buffer)**.

```
#include <android/log.h>

int main(){
__android_log_print(ANDROID_LOG_INFO, "SALUDO:", "%s", "Hola
mundo!");
}
```

Función __android_log_print(), para imprimir en consola

Uso de *viewmodels*

Según Android: “Se diseñó la clase ***ViewModel*** a fin de almacenar y administrar datos relacionados con la IU de manera optimizada para los ciclos de vida. La clase ***ViewModel*** permite que se conserven los datos luego de cambios de configuración, como las rotaciones de pantallas.” En otras palabras, esta clase sirve como intermediaria entre la vista y los datos de la aplicación, dando una mayor flexibilidad para el manejo de estos y su posterior representación en la IU. Su uso es recomendado por Android (ver “Recomendaciones para la arquitectura de Android”). También puede encontrarse cómo patrón MVVM (*Model-View-ViewModel*).

Hay dos situaciones que permiten ver el potencial de esta clase.

1. Los controladores de IU como las actividades y los fragmentos tienen un ciclo de vida. Este ciclo de vida es administrado por el *framework* de Android, el cual puede decidir destruir o recrear controladores. Si los datos son almacenados dentro de estos controladores, cada vez que se destruya uno, los datos que contenía se perderan.
2. Es necesario realizar operaciones asincrónicas, que podrían tardar en devolver.

Por lo tanto, es más fácil quitarle la propiedad de los datos a los controladores y manejarlos desde ***ViewModel***. Se puede ver un ejemplo práctico en “Cómo ejecutar tareas de Android en subprocesos en segundo plano**”**.

Address already in use

Durante el diseño de la aplicación se consideró que sólo se realizaría una comunicación entre un cliente y el servidor. Luego de intercambiar un mensaje cada uno, la conexión termina y el socket se cierra.

Cada vez que se establece una conexión, el puerto no puede volverse a utilizar inmediatamente. El motivo por el que esto es así está relacionado con el estado **TIME_WAIT**.

Es posible modificar el comportamiento del estado **TIME_WAIT**. Por ejemplo, podemos usar la opción **SO_REUSEADDR** para permitir que se pueda reutilizar el socket inmediatamente. También es posible reducir el tiempo de espera para este estado o manipular el comportamiento de este estado desde el sistema operativo.

De todas formas, no es recomendable manipular el estado **TIME_WAIT**, sino más bien es mejor adaptar la implementación para que pueda coexistir con este estado.

Mejoras propuestas

Un inconveniente que presenta la solución propuesta es que será necesario iniciar el servidor de nuevo cada vez que quiera establecerse una comunicación. Se podría utilizar el mismo servidor para escuchar múltiples conexiones o para enviar múltiples mensajes. Estas características no fueron consideradas a la hora del desarrollo.

Conclusión

Al igual que en GNU/Linux, se pudo comprobar que es posible utilizar sockets mediante las librerías nativas de C en Android. No fue necesario realizar cambios drásticos en el código para lograr su correcto funcionamiento.

En este documento se recorrió el proceso de desarrollo de aplicaciones para Android. Se mencionaron las herramientas más importantes para poder crear las aplicaciones, y se puso énfasis en JNI por ser la intermediaria entre el código Java y el código nativo. Para introducir al lector en el uso de JNI se propuso un ejemplo (basado en una plantilla que provee Android para el desarrollo de aplicaciones nativas). Seguidamente, se presenta el desarrollo de la aplicación que implementa un cliente y un servidor. La misma se presenta en tres etapas, donde se hace hincapié en distintos aspectos que influyen en el desarrollo del programa. Finalmente, se explican algunos conceptos adicionales y se proponen mejoras para la aplicación resultante.

Bibliografía

GNU C Library **- Capítulo 16. (s.f.)**
https://www.gnu.org/software/libc/manual/html_node/Sockets.html

Interfaz Nativa de Java (JNI). (s.f.). En Documentación de Java SE 6
https://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/intro.html#wp9502

Sitio web para el desarrollador Android, Bibliotecas Nativas
https://developer.android.com/training/articles/perf-jni?hl=es-419#native-libraries

Documentación de JNI Oracle (Capítulo 2): principales problemas de diseño del JNI, relacionados con los métodos nativos
https://docs.oracle.com/javase/6/docs/technotes/guides/jni/spec/design.html#wp615

JNI: Interfaz Nativa de Java. (2021). En Desarrolladores de Android
https://developer.android.com/training/articles/perf-jni?hl=es-419#general-tips

***Bionic*:**
Código fuente y documentación oficial de la librería
https://android.googlesource.com/platform/bionic/

Bionic **en Wikipedia**
https://en.wikipedia.org/wiki/Bionic_(software)

Procesos e hilos en Android:
Descripción general de los procesos y subprocesos
https://developer.android.com/guide/components/processes-and-threads?hl=es-419#Threads

Trabajo asíncrono con subprocesos de Java en Android
https://developer.android.com/guide/background/asynchronous/java-threads

Documentación de la interfaz *ExecutorService*
https://developer.android.com/reference/java/util/concurrent/ExecutorService

Clase *Service* Android
https://developer.android.com/guide/components/services?hl=es-419

Cómo ejecutar tareas de Android en subprocesos en segundo plano
https://developer.android.com/guide/background/threading?hl=es-419

Clase *ViewModel*
https://developer.android.com/topic/libraries/architecture/viewmodel?hl=es-419

Recomendaciones para la arquitectura de Android
https://developer.android.com/topic/architecture/recommendations?hl=es-419

Cómo enviar comandos de la consola del emulador
https://developer.android.com/studio/run/emulator-console?hl=es-419

Cómo configurar las redes del emulador de Android
https://developer.android.com/studio/run/emulator-networking?hl=es-419#networkaddresses

Comando mkfifo
https://www.gnu.org/software/coreutils/manual/html_node/mkfifo-invocation.html#mkfifo-invocation

TIME_WAIT *and its design implications for protocols and scalable client server systems*
https://serverframework.com/asynchronousevents/2011/01/time-wait-and-its-design-implications-for-protocols-and-scalable-servers.html

Linux IPv4 protocol implementation
https://man7.org/linux/man-pages/man7/ip.7.html

Bibliografía adicional

***Syscalls* BIONIC**
https://android.googlesource.com/platform/bionic/+/refs/heads/master/libc/SYSCALLS.TXT