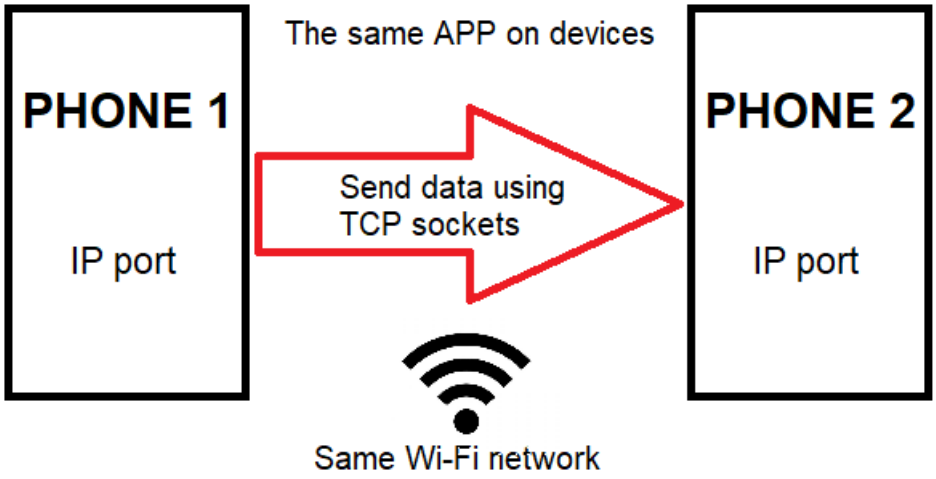
Java & Android w Android Studio (3.5.1)

Komunikowanie się urządzeń przez Wi-Fi

Tym razem zajmiemy się komunikacją urządzeń z systemem Android. Utworzymy aplikację, która będzie prostym komunikatorem Wi-Fi umożliwiającym przesyłanie wiadomości tekstowych pomiędzy dwoma smartfonami w architekturze klient-serwer. Komunikacja urządzeń będzie możliwa wewnątrz tej samej sieci przy wykorzystaniu gniazd TCP (TCP sockets).



Schemat działania aplikacji

Programowanie klient-serwer

Programowanie klient-serwer jest jedną z ważniejszych ogólnych i uniwersalnych metodologii budowy systemów informatycznych. Zyskała ona na znaczeniu w dobie rozwoju sieci komputerowych i dzisiaj jest praktycznie wszechobecna.

Architektura klient-serwer

Architektura klient-serwer jest technologią budowy systemów informatycznych, polegającą na podziale systemu na współdziałające ze sobą dwie kategorie programów lub procesów: klientów i serwerów.

Klient to program lub proces, który - oprócz wykonywania swoistych dla niego działań - łączy się z innym programem lub procesem, zwanym serwerem i poprzez kanały komunikacyjne zleca mu wykonanie określonych działań; w szczególności dostarczenia jakichś danych lub wyników przetwarzania danych.

Serwer jest programem lub procesem, który - na zlecenie klientów - świadczy określone usługi - np. dostarcza im dane lub wyniki przetwarzania jakichś danych.

Należy zwrócić uwagę na to, że pojęcie "klient-serwer" jest natury software'owej, a nie hardware'owej. Dotyczy programów, a nawet ich części (podprocesów, wątków), a nie sprzętu. Typowym błędem, popełnianym nie tylko w popularnej literaturze informatycznej, jest identyfikowanie pojęcia serwer z komputerem i traktowanie architektury klient-serwer jako wyłącznie sposobu komunikacji pomiędzy różnymi komputerami w sieci.

Tymczasem koncepcję klient-serwer możemy zastosować do budowy systemu informatycznego działającego na jednym komputerze w jednym systemie operacyjnym, a nawet dla budowy systemu, składającego się z jednego programu, w którym określony wątek będzie pełnił role serwera, a inne wątki będą zlecać mu różne usługi do wykonania.

Gniazda

Komunikacja pomiędzy procesami w sieci może odbywać się za pomocą jednego z wysokopoziomowych protokołów sieciowych (HTTP, FTP itp.), a także bardziej zaawansowanych mechanizmów takich jak zdalne wywołanie metod (RMI, CORBA), zdalne wywołanie procedur (RPC), czy SOAP (pozwalający na zdalne wolanie metod i procedur - inaczej: zlecanie usług - za pomocą przekazywania odpowiednio ustrukturyzowanych plików XML z zapisem "zadań do wykonania").

U podstaw wymiany informacji za pomocą protokołów najwyższego poziomu leży obecnie (najczęściej) komunikacja za pomocą gniazd.

Gniazdo (socket) - to abstrakcja programistyczna, oznaczająca punkt docelowy dwustronnej komunikacji dwóch procesów działających równolegle w sieci.

Gniazda są swoistymi kanałami komunikacyjnymi. Najbardziej naturalne było uczynienie komunikacji za pomocą gniazd w idei podobnej do dobrze znanego programistom paradygmatu operacji wejścia-wyjścia. Zatem mamy tu naturalną sekwencję: otwarcie kanału komunikacji (gniazda), zapis lub odczyt (przesłanie lub otrzymanie danych za pomocą gniazda), usunięcie kanału komunikacyjnego (zamkniecie gniazda).

Zwykle komunikacja za pomocą gniazd implementowane jest na bazie protokołu TCP lub protokołu UDP.

Protokół TCP (Transport Control Protocol) jest protokołem połączeniowym (co znaczy, że ustanawiana jest dwustronne połączenie pomiędzy klientem i serwerem). Zapewnia, że dane posyłane poprzez gniazda docierają w całości i w odpowiedniej kolejności. Inaczej możemy powiedzieć, że realizowana jest tu strumieniowa koncepcja wymiany danych, co oznacza, że po ustanowieniu połączenia można przesłać dane o dowolnym rozmiarze i operacje wymiany danych możemy wykonywać tutaj za pomocą dobrze nam znanych strumieni.

Protokół **UDP** (User Datagrams Protocol) jest protokołem bezpołączeniowym. Dane przesyłane są pomiędzy procesami jako datagramy (pakiety danych o określonej maksymalnej wielkości np. 64 kB), przy czym z każdym datagramem posyłany jest "adres" odbiorcy. Datagramy mogą więc przybywać na miejsce przeznaczenia (do innego procesu) w dowolnej kolejności (a niektóre nawet mogą w ogóle nie dotrzeć).

Oba protokoły (TCP i UDP) są protokołami typu "point-to-point", czyli każdorazowo zapewniającymi komunikację tylko pomiędzy dwoma procesami (w szczególności na dwóch różnych maszynach w sieci). Istnieje również możliwość użycia tzw. multicastingu. Ten rodzaj protokołów oznacza dystrybucje informacji z serwera od razu do wielu klientów. Oparty jest on na protokole UDP.

Identyfikacji maszyn biorących udział w komunikacji (tzw. hostów) służy **protokół IP** (Internet Protocol). Adresy IP mają ogólnie formę 32-bitowych (lub 128-bitowych w wersji IPv6) liczb i mogą być zapisywane jako sekwencja czterech (ośmiu) liczb rozdzielonych kropkami (np. 192.33.71.12).

Identyfikacja hosta jest jednak nie wystarczająca dla komunikacji miedzy procesami: na danym komputerze może się przecież wykonywać równolegle wiele procesów.

Po to, by dane dotarły do określonego procesu protokoły TCP i UDP posługują się tzw. portami.

Porty są identyfikowane przez 16-bitowe liczby - numery portów. Numery te są używane przez TCP lub UDP do przesyłanie danych do odpowiedniego procesu.

W protokołach połączeniowych (takich jak TCP) proces-serwer przydziela sobie port o określonym numerze i poprzez ten właśnie port procesy klienckie mogą ustanawiać połączenia z serwerem.

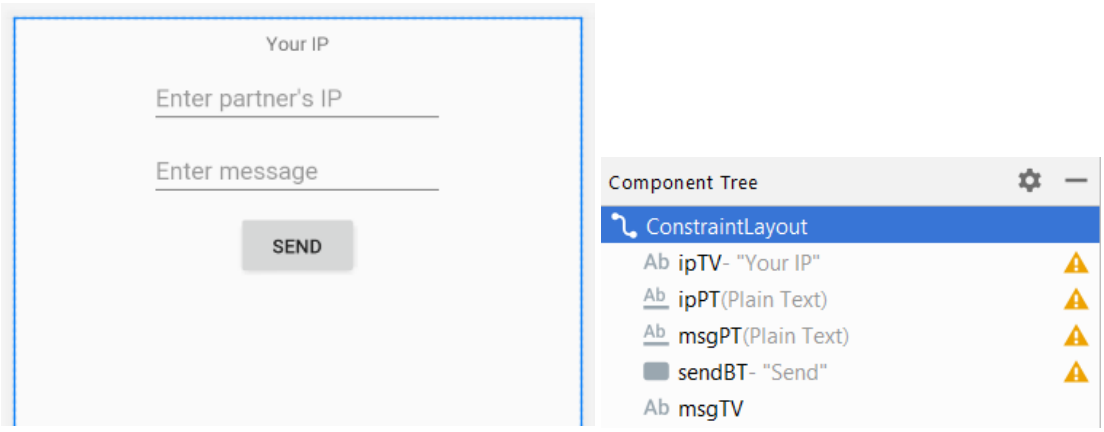
Tworzenie aplikacji wykorzystującej komunikację Wi-Fi

Idea naszej aplikacji polegać będzie na wprowadzeniu adresu IP rozmówcy oraz treści wiadomości, po czym wysłanie jej za pomocą kliknięcia przycisku. Oprócz tego w dolnej części wyświetlana będzie całą rozmowa, tzn. wszystkie wysłane i odebrane wiadomości. W naszej aplikacji umieścimy też informację o adresie IP naszego urządzenia.

Przygotujmy layout naszej aplikacji.

ZADANIE

Utwórz nowy projekt Android Studio w języku Java o nazwie **SocketsApp**, zawierający **EmptyActivity**. W pliku layoutu głównej aktywności użyj komponentu **ConstraintLayout** (domyślnie ustawiony). Dodaj do layoutu komponenty:
- **TextView** (id=ipTV, marginTop=8dp, text=”Your IP”),
- poniżej 2x **PlainText** (id=ipPT, marginTop=8dp, text=””, hint=”Enter partner's IP” oraz id=msgPT, marginTop=8dp, text=””, hint=”Enter message”),
- niżej **Button** (id=sendBT, marginTop=8dp) oraz
- kolejny **TextView** (id=msgTV, marginTop=8dp, text=””, height=65dp, gravity=bottom, scrollbars=vertical).



Layout naszej aplikacji jest gotowy, przystępujemy do oprogramowania funkcji. Na początek dodajmy w pliku manifestu stosowne uprawnienia dla naszej aplikacji, tj. możliwość korzystania z komunikacji internetowej oraz dostępu do parametrów sieci (połączenia).

ZADANIE

Dodaj w pliku manifestu poniższe uprawnienia.

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

Utwórzmy teraz zmienne reprezentujące komponenty layoutu w celu wygodniejszego dostępu do nich.

ZADANIE

Zaimplementuj poniższy kod klasy głównej aktywności.

```
public class MainActivity extends AppCompatActivity {

    TextView ipTV, msgTV;
    EditText ipET, msgET;

    @Override
    protected void onCreate(Bundle savedInstanceState) {
        super.onCreate(savedInstanceState);
        setContentView(R.layout.activity_main);

        ipTV = (TextView) findViewById(R.id.ipTV);
        ipET = (EditText)findViewById(R.id.ipPT);
        msgET = (EditText)findViewById(R.id.msgPT);
        msgTV = (TextView) findViewById(R.id.msgTV);
        msgTV.setMovementMethod(new ScrollingMovementMethod()); // scrolling
```

Ostatnia linia kodu (wraz z wcześniejszymi ustawieniami parametrów komponentu) odpowiada za automatyczne przewijanie tekstu w **TextView** wyświetlającym treść rozmowy.

Dodajmy funkcję zwracającą adres IP naszego smartfona.

ZADANIE

Dodaj poniższą funkcję w klasie **MainActivity**.

```
public String getIPAddr() {
    try {
        // uchwyt do managera sieci WiFi
        WifiManager wm = (WifiManager) getApplicationContext().getSystemService(WIFI_SERVICE);
        // pobranie adresu IPv4 urządzenia
        String myIP = Formatter.formatIpAddress(wm.getConnectionInfo().getIpAddress());
        return myIP;
    } catch (Exception ignored) { }
    return "";
}
```

Korzystając z utworzonej metody wyświetl adres IP urządzenia w komponencie **TextView**.

```
ipTV.setText("Your IP: "+getIPAddr());
```

Przystępujemy do oprogramowania przycisku do wysyłania wiadomości. W systemie Android operacje sieciowe nie mogą być bezpośrednio wykonywane w wątku interfejsu aplikacji, dlatego utworzymy klasę umożliwiającą wykonywanie w tle operacji związanych z wysyłaniem wiadomości wykorzystując klasę **AsyncTask**.

AsyncTask umożliwia prawidłowe i łatwe korzystanie z wątku interfejsu użytkownika. Ta klasa pozwala na wykonywanie operacji w tle i publikowanie wyników w wątku interfejsu użytkownika bez konieczności manipulowania wątkami i / lub programami obsługi.

Zadanie asynchroniczne jest definiowane przez obliczenia działające w wątku w tle i których wynik jest publikowany w wątku interfejsu użytkownika. Zadanie asynchroniczne jest definiowane przez 3 typy generyczne o nazwiach: Params, Progress i Result, oraz 4 kroki: onPreExecute, doInBackground, onProgressUpdate i onPostExecute.

ZADANIE

Zaimplementuj klasę daną niżej wewnątrz **MainActivity** (jako klasa wewnętrzna).

```
class BackgroundClass extends AsyncTask<String, Void, Void> {
    Socket s;
    DataOutputStream dos;
    String ip, message;
    @Override
    protected Void doInBackground(String... params) {
        ip = params[0];
        message = params[1];
        try{
            s = new Socket(ip, port: 9700);
            dos = new DataOutputStream(s.getOutputStream());
            dos.writeUTF(message);
            dos.close();
            s.close();
        }catch (IOException ioe){
            ioe.printStackTrace();
        }
        return null;
    }
}
```

W naszym przypadku **AsyncTask** korzysta jedynie z parametrów typu String (Progress=Void, Result=Void).

Do metody **doInBackground()** przekazujemy adres IP rozmówcy i treść wiadomości do wysłania. Następnie tworzone jest nowe gniazdo (połączenie) z wybranym adresem na porcie 9700 oraz strumień wyjściowy. Za pomocą strumienia przesyłamy wiadomość, po czym zamykamy strumień i połączenie.

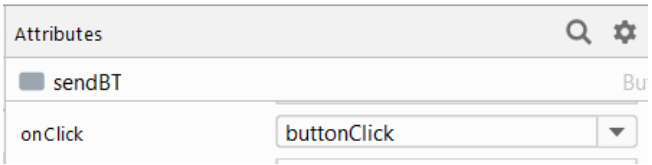
Utwórzmy teraz funkcję, która uruchamiać będzie asynchroniczne zadanie wysyłania wiadomości.

ZADANIE

Zaimplementuj w **MainActivity** funkcję:

```
public void buttonClick(View v) {
    BackgroundClass bc = new BackgroundClass();
    bc.execute(ipET.getText().toString(), msgET.getText().toString());
    msgTV.append("Message sended to client: "+msgET.getText().toString()+"\n");
}
```

Ustaw wywoływanie utworzonej funkcji w momencie kliknięcie przycisku **send**. W tym celu ustaw wartość atrybutu **onClick** w następujący sposób:



W funkcji **buttonClick()** uruchamiamy zadanie wysyłania wiadomości podając jako parametry adres IP rozmówcy i treść wiadomości pobrane z odpowiednich pól tekstowych. Dalej wyświetlamy treść wysłanej wiadomości w komponencie **TextView** z rozmową.

Mamy już gotową część kodu dla klienta (wysyłanie wiadomości), teraz musimy jeszcze oprogramować stronę serwera, który będzie odczytywał odebrane wiadomości i wyświetlał je w komponencie z treścią rozmowy.

Kod serwera musi działać na osobnym wątku, aby ciągle „nasłuchiwać” na danym porcie. Dlatego utworzymy dedykowaną mu klasę implementującą interfejs **Runnable**.

ZADANIE

Zaimplementuj klasę serwera wewnątrz **MainActivity** (jako klasa wewnętrzna).

```
class MyServer implements Runnable{
    ServerSocket ss;
    Socket mySocket;
    DataInputStream dis;
    String message;
    Handler handler = new Handler();

    @Override
    public void run() {
        try {
            ss = new ServerSocket( port: 9700);
            handler.post(() → {
                    Toast.makeText(getApplicationContext(), text: "Waiting for client",
                            Toast.LENGTH_LONG).show();
            });
            while(true){
                mySocket = ss.accept();
                dis = new DataInputStream(mySocket.getInputStream());
                message = dis.readUTF();
                handler.post(() → {
                        msgTV.append("Message recived from client: "+message+"\n");
                });
            }
        }catch (IOException ioe){
            ioe.printStackTrace();
        }
    }
}
```

W metodzie **run()** tworzymy (otwieramy) nowe gniazdo serwera na porcie 9700 (zgodny z portem klienta). Dalej w pętli nieskończonej nasłuchujemy na połączenia przychodzące (ss.accept()). Następnie Tworzymy strumień wejściowy (przychodzący) i odczytujemy wiadomość od klienta.

Teraz pozostaje jeszcze wywołać wątek serwera w metodzie **onCreate()**.

ZADANIE

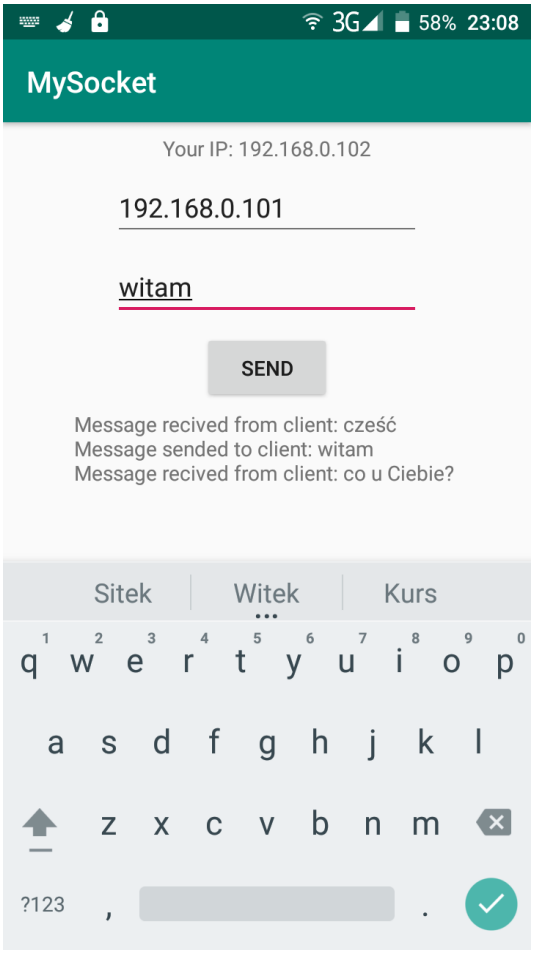
Wywołaj wątek serwera w metodzie **onCreate()** głównej aktywności.

```
Thread myThread = new Thread(new MyServer());
myThread.start();
```

Uruchom aplikację i przetestuj jej działanie. Pamiętaj o wcześniejszym włączeniu modułu Wi-Fi i połączeniu się z siecią (klient i serwer muszą być w tej samej sieci).

Kod naszej aplikacji jest już gotowy. Poniżej zrzuty ekranu działającej aplikacji.


Tylko połączeni... 89% 23:08
MySocket
Your IP: 192.168.0.101
192.168.0.102
co u Ciebie?
SEND
Message sended to client: cześć
Message recived from client: witam
Message sended to client: co u Ciebie?
Gotowe


3G 58% 23:08
MySocket
Your IP: 192.168.0.102
192.168.0.101
witam
SEND
Message recived from client: cześć
Message sended to client: witam
Message recived from client: co u Ciebie?
Sitek
Witek
Kurs