



# Lezione 22



# Programmazione Android



- Tecnologie di rete
  - Networking TCP/IP
  - Il Connectivity manager
  - Bluetooth
  - Wi-fi direct



# Networking TCP/IP



# Visione generale



- Il networking via TCP/IP su Android è **completamente standard**
- Vale tutto quanto conoscete sul networking in Java
- Il S.O. Utilizza tutti i “trucchi” soliti per garantire (best-effort) il delivery dei vostri pacchetti
  - Routing su reti di diverso tipo (wi-fi o cellulari)
  - Gestione dinamica al variare della connessione



# Esempio

ovvero

## Networking in Java in 2 slide



- Server-side

```
try {  
    ServerSocket ss = new ServerSocket(8080);  
    while (!done) {  
        Socket s = ss.accept();  
        servi(s);  
    }  
} catch (...) { ... }
```

- Client-side

```
try {  
    Socket s = new Socket(server, 8080);  
    ordina(s);  
} catch (...) { ... }
```



# Esempio

ovvero

## Networking in Java in 2 slide



- Letture e scritture su rete avvengono tramite gli *stream* tipici di Java
  - Dal Socket si estraggono `InputStream` e `OutputStream`

```
InputStream is = s.getInputStream();  
OutputStream os = s.getOutputStream();
```
  - Letture e scritture avvengono come normale
    - Tipicamente, incapsulando gli stream in `BufferedStream`, `PrintWriter`, `DataInput/OutputStream`, `ObjectInput/OutputStream`, ecc.



# Concorrenza

- Naturalmente, le operazioni su rete possono essere lente e/o bloccanti
  - **Mai** accedere alla rete nel thread UI!
- È anche poco comune avere un server sul cellulare
  - Il server dovrebbe, per sua natura, essere in esecuzione *sempre...*
  - L'approccio di Android è che i programmi utente dovrebbero essere in esecuzione *mai...*
    - Per ricevere notifiche push, si usano altre strade

# Batteria

- Le operazioni su rete consumano in genere molta energia (specialmente in trasmissione)
  - Necessaria perché i segnali radio siano ricevuti a destinazione
- È opportuno cercare di minimizzare il consumo
  - Trasferire pochi dati (aiuta anche con i costi)
  - Mantere delle cache quando possibile
  - Fare *coalescing*: poche comunicazioni “grosse”
    - Ogni ri-accensione dei modem 3G richiede tempo e energia
    - Se dovete fare una comunicazione, fatene tante insieme!



# Il ConnectivityManager

# ConnectivityManager

- Il ruolo del ConnectivityManager è di fornire informazioni sulle reti accessibili al dispositivo
  - Momento per momento

- Si tratta dell'ennesimo servizio di sistema:

```
ConnectivityManager cm = (ConnectivityManager)  
    getSystemService(Context.CONNECTIVITY_SERVICE);
```

- Ogni network è rappresentato da un oggetto

```
NetworkInfo[] ni = cm.getAllNetworkInfo();
```

```
NetworkInfo currni = cm.getActiveNetworkInfo();
```

```
NetworkInfo wfni = cm.getNetworkInfo(NetworkInfo.TYPE_WIFI);
```



# NetworkInfo

- Un oggetto NetworkInfo mantiene tutte le informazioni disponibili su un particolare network
- Tipi supportati:
  - TYPE\_BLUETOOTH
  - TYPE\_DUMMY
  - TYPE\_ETHERNET
  - TYPE\_MOBILE
    - TYPE\_MOBILE\_DUN, TYPE\_MOBILE\_HIPRI, TYPE\_MOBILE\_MMS, TYPE\_MOBILE\_SUPL
  - TYPE\_VPN
  - TYPE\_WIFI
  - TYPE\_WIMAX

API 21+

Nota: negli US, Google è appena diventato un VNO (virtual network operator), e ha annunciato di voler instradare le chiamate vocali sulle sue reti VPN...



# NetworkInfo

- Una volta ottenuto un NetworkInfo, si possono chiedere le caratteristiche di dettaglio della rete
  - Info: getType(), getTypeName(), getDetailedState(), getExtraInfo()
  - Stato: isAvailable(), isConnected(), isFailOver(), isRoaming() isConnectedOrConnecting(),
  - Errori: getReason()



# Notifiche



- Il ConnectionManager invia degli Intent broadcast ogni volta che cambiano le condizioni della rete
  - Action CONNECTIVITY\_ACTION
  - Extras contiene ulteriori info
- Potete registrare un BroadcastReceiver nel manifest, con un intent filter
  - Ma in questo caso, il vostro codice verrà eseguito **sempre**, anche quando l'app non è in esecuzione
- Oppure, registrare dinamicamente il receiver
  - Di solito, in onCreate() / onDestroy()



# Uso delle notifiche per ottimizzare la batteria



- Un'applicazione può ricevere notifiche che segnalano il passaggio a un nuovo tipo di rete
  - Per esempio, fra wi-fi e cellulare
- In molti casi, è utile modificare i pattern di accesso alla rete in base al tipo (e alle preferenze)
  - Per esempio, scaricare un podcast solo su wi-fi, ma ricevere le notifiche sulla disponibilità di una nuova puntata anche su 3G
- Nei casi di polling, ricordarsi degli inexact alarms
  - Si può fare coalescing anche fra app diverse!



# Bluetooth

# Bluetooth

- Bluetooth è la più diffusa tecnologia per le *personal area network*
  - L'idea è di offrire connettività solo a dispositivi che sono in prossimità fisica alla persona dell'utente
  - Protocollo complicato, prevede molti *profili* per diverse classi di dispositivi
    - Auricolari, trasferimento file, streaming di audio in casa, dispositivi medicali, ecc.
- Ci interessa in particolare il profilo RFCOMM
  - Comunicazioni seriali in radiofrequenza

# Pairing & co.

- Bluetooth prevede che i dispositivi debbano essere *accoppiati* prima di poter scambiare dati a livello applicativo
  - Naturalmente, possono scambiare dati anche prima (per esempio, i loro nomi e classi), ma solo a livello di protocollo
- L'accoppiamento deve confermare la volontà dell'utente/proprietario di entrambi i dispositivi
  - Per questo motivo, richiede in genere un segnale esplicito da parte sua: passkey (PIN)

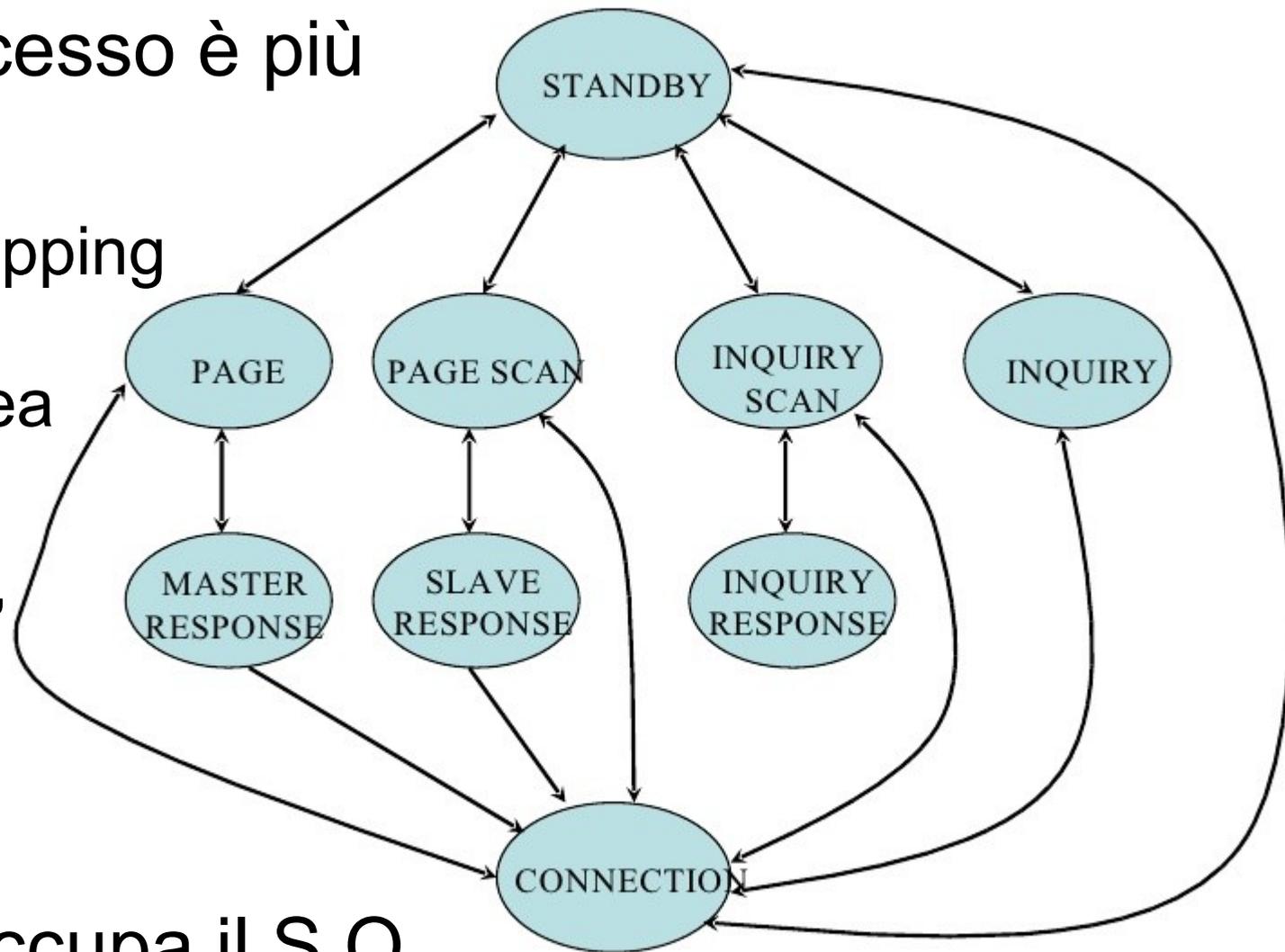
# Pairing & co.

- Un dispositivo Bluetooth può quindi essere in diversi stati
  - Spento: modulo Bluetooth non attivato
  - Acceso, ma non discoverable
  - Discoverable ma non paired
  - Discoverable e paired
  - Paired e non discoverable
- D'altra parte, l'altro dispositivo può attivamente cercare dispositivi discoverable o no

# Pairing & co.



- In realtà il processo è più complesso
  - Frequency hopping su 12 canali diversi per area geografica
  - Point-to-point, piconet, scatternet
  - Master/slave
- ... ma se ne occupa il S.O.



# Le classi principali

- Nella maggior parte dei casi, si usano 4 classi
  - **BluetoothAdapter** – rappresenta la scheda di rete
  - **BluetoothDevice** – rappresenta un dispositivo
  - **BluetoothServerSocket** e **BluetoothSocket**
    - Analoghi a ServerSocket e Socket del networking TCP/IP
- Esistono poi diverse altre classi
  - Specializzazioni per particolari profili
  - Classi che descrivono i meta-dati dei profili
  - Non le vedremo...

# BluetoothAdapter

- In teoria, un dispositivo potrebbe avere più adapter Bluetooth; occorre quindi prendere una particolare istanza, per esempio:

```
BluetoothAdapter bta =BluetoothAdapter.getDefaultAdapter();
```

- Se **bta** è null, il dispositivo non supporta BT
- Altrimenti, si controlla se BT è abilitato, e in caso contrario si chiede all'utente di abilitarlo

```
if (!bta.isEnabled()) {  
    Intent i=new Intent(BluetoothAdapter.ACTION_REQUEST_ENABLE);  
    startActivityForResult(i, 1);  
}
```

# Abilitare Bluetooth

- L'utente a questo punto è libero di decidere se abilitare o meno BT
- L'activity che risponde all'intent è un dialog di sistema
- Verrà poi chiamata la `onActivityResult()` passando un risultato che indica l'esito



```
void onActivityResult(int code, int res, Intent data) {  
    If (code==1) {  
        If (res==RESULT_OK) { /* BT abilitato! */ }  
        if (res==RESULT_CANCEL) { /* niet! */ }  
    } ...  
}
```



# Abilitare Bluetooth



- In alternativa, è anche possibile registrarsi per ricevere l'intent broadcast con action `BluetoothAdapter.ACTION_STATE_CHANGED`
- Gli extra dell'intent contengono informazioni sullo stato corrente
  - L'app può anche fare “piggybacking” – *non* richiede lei di attivare il BT, ma è pronta a partire se qualcun altro lo attiva
  - Stessa tecnica per scoprire la disattivazione



# Discovery



- Il secondo passo consiste nello scoprire con quali BluetoothDevice possiamo comunicare
- Dispositivi paired in passato (e registrati):  

```
Set<BluetoothDevice> devs = bta.getBondedDevices();
```
- Dispositivi discoverable e in range:  

```
IntentFilter f = new IntentFilter(BluetoothDevice.ACTION_FOUND);  
registerReceiver(this, f);  
bta.startDiscovery();
```

  - Il nostro onReceive() riceverà tanti intent ACTION\_FOUND quanti sono i dispositivi in range
    - Ciascuno ha negli extra una descrizione completa



# Discovery



- Esempio di receiver:

```
void onReceive(Context c, Intent i) {  
    String action = i.getAction();  
    if (BluetoothDevice.ACTION_FOUND.equals(action)) {  
        BluetoothDevice dev =  
            i.getParcelableExtra(BluetoothDevice.EXTRA_DEVICE);  
    } ...  
}
```

- Per rendersi discoverable a propria volta:

```
Intent i=new Intent(BluetoothAdapter.ACTION_REQUEST_DISCOVERABLE);  
startActivityForResult(i, 2);
```

- Vale quanto detto per l'abilitazione (RESULT\_OK/CANCELED)



# Collegamento

- Una volta che sia stato effettuato il pairing, i due dispositivi possono comunicare
- Uno farà da server, l'altro da client
  - Il client è l'iniziatore della connessione
  - Il server si mette in attesa di connessione
- La connessione è ammessa solo se **entrambi i device presentano lo stesso UUID**
  - L'UUID può essere un accordo privato
  - Svolge lo stesso ruolo del numero di porta in TCP/IP



# Collegamento – server



```
String myname = "it.unipi.di.sam.bttest server";  
UUID myid = UUID.fromString("550e8400-e29b-41d4-a716-446655440000");  
BluetoothServerSocket bss =  
    bta.listenUsingRfcommWithServiceRecord(myname, myid);  
BluetoothSocket bs = bss.accept();  
bss.close();  
servi(bs);
```

- L'intero processo è simile a quello per TCP/IP
  - Tuttavia, raramente il server rimane attivo in attesa di ulteriori connessioni; più spesso si tratta di connessioni singole
  - Come al solito: mai nel thread UI!



# Collegamento – client



- Il client deve specificare a quale device e servizio vuole connettersi
  - Il device è uno dei device paired (ottenuto come visto prima)
  - Il servizio è identificato dall'UUID

```
UUID hisid = UUID.fromString("550e8400-e29b-41d4-a716-446655440000");  
BluetoothDevice btd = ... ;  
BluetoothSocket bs = btd.createRfcommSocketToServiceRecord(hisid) ;  
bs.connect(); /* ← exception se la connessione non riesce */  
ordina(bs) ;
```



# Scambio di dati

- Una volta che sia il server che il client hanno ottenuto un loro `BluetoothSocket`, il processo è il solito
  - Si ottengono `InputStream` e `OutputStream` per ogni socket
  - Direttamente o tramite wrapping si procede allo scambio di dati
  - Problemi di rete si traducono in fallimenti delle letture/scritture o eccezioni
  - Al termine, si chiama `close()` per chiudere la `RFCOMM`



# Wi-fi direct



# Wi-fi direct

- Wi-fi direct è una tecnologia relativamente nuova per le connessioni point-to-point via wi-fi
  - Disponibile da Android 4.0 in poi
  - Casi d'uso analoghi a Bluetooth, ma molto più veloce e con range assai più esteso
  - I dispositivi devono avere entrambi wi-fi, ma **non** necessitano di una base station
- Consente anche di creare **gruppi p2p**
  - Dispositivi multipli, tutti in range, che comunicano liberamente



# Design del framework



- A differenza dei network layer precedenti, lo strato wi-fi direct di Android è stato sviluppato nativamente
- Fa quindi uso del design più tipico di Android
  - Si lanciano intent per chiedere azioni
  - Si ricevono notifiche via listener
- La classe WifiP2pManager fornisce i metodi che semplificano la vita

In effetti, il nome originale della tecnologia era “Wi-Fi P2P”, quando il P2P era cool. Quando sono cominciate le cause contro i pirati del P2P, il nome “WiFi Direct” improvvisamente suonava meglio...



# WifiP2pManager



- Il solito servizio di sistema

```
WifiP2pManager wfdm = (WifiP2pManager)  
    getSystemService(Context.WIFI_P2P_SERVICE);
```

- L'inizializzazione del sistema ci fornisce un canale da usare poi come handle

```
Channel ch = wfdm.initialize();
```

- I metodi del WifiP2pManager consentono di chiedere le principali funzioni

- Le risposte arriveranno tramite intent broadcast o attraverso chiamate a dei listener



# WifiP2pManager



- **Metodi principali:**

Method	Description
initialize()	Registers the application with the Wi-Fi framework. This must be called before calling any other Wi-Fi Direct method.
connect()	Starts a peer-to-peer connection with a device with the specified configuration.
cancelConnect()	Cancels any ongoing peer-to-peer group negotiation.
requestConnectInfo()	Requests a device's connection information.
createGroup()	Creates a peer-to-peer group with the current device as the group owner.
removeGroup()	Removes the current peer-to-peer group.
requestGroupInfo()	Requests peer-to-peer group information.
discoverPeers()	Initiates peer discovery
requestPeers()	Requests the current list of discovered peers.

# Discovery



- Il processo di Discovery inizia con una chiamata a `discoverPeers()`:

```
wfdm.discoverPeers(ch, new WifiP2pManager.ActionListener()
{
    @Override
    public void onSuccess() {
        ...
    }

    @Override
    public void onFailure(int reasonCode) {
        ...
    }
});
```

Indica solo che è andata bene

Codice di fallimento

# Discovery



- Una volta che il processo di discovery è concluso, il sistema invia un broadcast intent
  - `WIFI_P2P_PEERS_CHANGED_ACTION`
- A questo punto, la nostra applicazione (che avrà “visto” l'intent tramite un receiver) può chiamare `requestPeers()`

```
void onReceive(Context c, Intent i) {  
    String action = i.getAction();  
    if (WifiP2pManager.WIFI_P2P_PEERS_CHANGED_ACTION.equals(action)) {  
        wfdm.requestPeers(ch, pllist);  
    } ...  
}
```

Altro listener!



# Discovery



- Quest'ultimo listener sarà un `WifiP2pManager.PeerListListener`
  - Unico metodo:  
`onPeersAvailable(WifiP2pDeviceList peers)`
- Finalmente, `peers` contiene la lista dei dispositivi raggiungibili!
- È ora possibile iniziare una connessione

# Connessione

- Assumendo che dev sia un dispositivo dalla lista, possiamo creare una **configurazione** corrispondente, ed effettuare una connessione

```
WifiP2pConfig cfg = new WifiP2pConfig();
cfg.deviceAddress = dev.deviceAddress;
wfdm.connect(ch, cfg, new ActionListener() {
    @Override
    public void onSuccess() {
        ...
    }
    @Override
    public void onFailure(int reason) {
        ...
    }
});
```



# Trasferimento

- La buona notizia:
  - Una volta che i dispositivi sono connessi, si possono usare le normali socket TCP/IP
  - Compaiono come dispositivi su una rete locale privata
  - Niente di nuovo da scoprire!



# NFC



# NFC



- All'altro estremo della gamma, abbiamo le comunicazioni NFC
  - NFC = near field communication
  - Distanza massima: 4cm (a contatto)
- Tre casi d'uso:
  - Lettura / scrittura di *tag NFC*
  - NFC P2P
  - Simulazione di un *tag NFC*



# Tag NFC



- I tag NFC fungono da “memoria di massa”
  - Capacità molto limitata
- Le applicazioni possono registrarsi per essere attivate quando il dispositivo è posto a contatto con un tag NFC

```
<intent-filter>  
  <action android:name="android.nfc.action.TAG_DISCOVERED"/>  
</intent-filter>
```
- L'intent contiene fra gli extra le informazioni del tag
  - Esistono alcuni altri intent per casi più specifici