Fuori programma
Google Wear APIs

Esempio: WearAPI

- Una volta ottenuto il `gapi` per Wear.API, si può fare accesso a cinque sotto-API, ciascuna espressa da una interfaccia Java
  - Replicazione e sincronizzazione
    - **DataApi** – legge e scrive DataItem e Asset, che vengono sincronizzati automaticamente fra device
  - Invio di messaggi fra nodi di una rete
    - **MessageApi** – invio di messaggi fra nodi
    - **NodeApi** – informazioni sui nodi disponibili sulla rete
    - **ChannelApi** – creazione di canali fra nodi
    - **CapabilityApi** – scopre le caratteristiche dei nodi sulla rete

DataAPI

- L'idea di fondo della DataAPI è di fornire uno **spazio di memorizzazione** virtuale, condiviso fra più dispositivi
  - Ogni volta che uno dei dispositivi modifica un elemento, il nuovo elemento viene replicato (sincronizzato) sugli altri dispositivi
    - Opzionalmente, l'applicazione viene notificata
  - Il sistema gestisce caching, buffering, trasmissione, ritrasmissione, perdite temporanee di connessione, ecc.
    - Approccio “eventually consistent”, largamente trasparente

DataAPI

DataItem A

DataItem B

Asset B

DataItem A

DataItem B

Asset B

DataItem A

DataItem B

Asset B

Da GooglePlayServices 7.3 (Maggio 2015), si possono connettere più Wearable allo stesso Mobile

DataItem

- Un **DataItem** è una unità di sincronizzazione
  - Immaginate: un file nello storage condiviso
- Dotato di:
  - Un **nome** simbolico (stringa nel formato dei pathname assoluti UNIX: “/voti/conteggio”)
  - Un **contenuto** arbitrario (array di byte)
    - Limite massimo: 100Kb
    - Può essere gestito a mano, oppure creato tramite una **DataMap** che lo incapsula in un bundle chiavi-valori

DataItem
Approccio diretto

- Se volete gestire il contenuto a mano…
- Creazione/scrittura
  - pdr=PutDataRequest.create(path).setData(buffer)
  - pr=DataApi.putDataItem(gapi,pdr)
  - pr sarà un PendingResult<DataApi.DataItemResult>
- Un DataItem è identificato da una URI
  - Accessibile con DataItem.getUri()
  - wear://*nodo*/*path* o wear:/*path*

Copia su un nodo specifico

Il dato in sé, su tutti i nodi

DataItem
Approccio diretto

- Per leggere il valore corrente di un DataItem
  - pr = DataApi.getDataItem(gapi, uri) oppure
    pr = DataApi.getDataItems(gapi)
  - pr sarà un PendingResult<DataItemBuffer>
  - Potete usare getCount() e get(idx) sul DataItemBuffer per enumerare i DataItem
  - Sul DataItem possiamo finalmente invocare getData() per recuperare il buffer di byte memorizzato

DataItem
Approccio con DataMap

- Per evitare di gestire a mano il buffer di dati, si può passare da una *ulteriore* classe detta DataMap
- pdmr = PutDataMapRequest.create(path);
- dm = pdmr.getDataMap();
- dm.put*Tipo*(chiave, valore); …
- pdr = pdmr.asPutDataRequest();
- Da qui si procede come prima per le PutDataRequest

DataItem
Approccio con DataMap

- Esempio: condividere nome ed età della nonna

```
private void condividi(String nome, int età) {
    PutDataMapRequest pdmr = PutDataMapRequest.create("/nonna");
    DataMap dm = pdrm.getDataMap();
    dm.putString(K_NOME, nome);
    dm.putInt(K_ETA, età);
    PutDataRequest pdr = pdmr.asPutDataRequest();
    PendingResult<DataApi.DataItemResult> pr =
                Wearable.DataApi.putDataItem(gapi, pdr);
}
```

- pr.await() (o una callback) per essere avvisati del termine dell'operazione
- Ma la sincronizzazione potrebbe avvenire dopo!

DataItem
Approccio con DataMap

- Per leggere il valore corrente
  - Si recupera il DataItem come visto in precedenza
  - Si ottiene il DataMap corrispondente con

```
DataMap dm = DataMapItem.fromDataItem(di).getDataMap();
```

  - Da dm si estraggono i dati con i soliti metodi “stile Bundle”
    - dm.get*Tipo*(chiave)
- DataMap offre anche metodi per convertire da Bundle a DataMap e viceversa

Gli Asset

- A un DataItem può essere associato zero o più **Asset**
- Blocchi di dati di grandi dimensioni
  - Non hanno il limite di 100Kb
- Vengono trasferiti insieme ai DataItem
  - Però il sistema adotta politiche di caching più aggressive per evitare di sovraccaricare la connessione BlueTooth
- Acceduti tramite file descriptor

Gli Asset

- Per costruire un Asset, si usa uno dei vari metodi statici create...() della classe Asset
  - createFromBytes(byte[] buffer)
  - createFromFd(ParcelFileDescriptor fd)
  - createFromRef(String digest)
  - createFromUri(Uri uri)
- Ottenuto l'Asset, lo si aggiunge al DataItem con una chiamata a putAsset(nome, asset)
  - Nome sarà il nome simbolico dell'asset dentro il DataItem

Gli Asset
Esempio: un'immagine

- Approccio diretto

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.avatar);
ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
bitmap.compress(Bitmap.CompressFormat.PNG, 100, byteStream);
Asset asset = Asset.createFromBytes(byteStream.toByteArray());
PutDataRequest pdr = PutDataRequest.create("/poster");
pdr.putAsset("locandina", asset);
PendingResult<DataApi.DataItemResult> pr = Wearable.DataApi.putDataItem(gapi, pdr);
```

- Approccio con DataMap

```
Bitmap bitmap = BitmapFactory.decodeResource(getResources(), R.drawable.avatar);
ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
bitmap.compress(Bitmap.CompressFormat.PNG, 100, byteStream);
Asset asset = Asset.createFromBytes(byteStream.toByteArray());
PutDataMapRequest pdmr = PutDataMapRequest.create("/poster");
pdmr.getDataMap().putAsset("locandina", asset)
PutDataRequest pdr = pdmr.asPutDataRequest();
PendingResult<DataApi.DataItemResult> pr = Wearable.DataApi.putDataItem(gapi, pdr);
```

Gli Asset

- Una volta recuperato il DataItem, si possono estrarre gli Asset con getAsset()
  - Chiamato su DataItem nel caso manuale
    - Con il nome appropriato
  - Chiamato su DataMap nel caso si usi DataMap
    - Con la chiave appropriata
- Finalmente, dall'Asset così recuperato possiamo estrarre i dati chiamando
  - getUri() – sotto forma di URI
  - getFd() – sotto forma di ParcelFileDescriptor

DataAPI: notifiche

- Naturalmente, nella maggior parte dei casi vorremo essere **notificati** quando una versione aggiornata di un DataItem è disponibile!
  - Perché un altro nodo ha modificato la sua copia
- Come al solito, possiamo registrare un Listener che verrà invocato quando necessario
  - L'interfaccia da implementare è DataApi.DataListener
  - Pattern solito: la si può fare implementare all'Activity

DataAPI: notifiche

- Solito schema
  - DataApi.registerListener(gapi,listener)
  - DataApi.removeListener(gapi,listener)
- Il listener ha un solo metodo:
  - public void onDataChanged(DataEventBuffer evb)
- evb offre i metodi getCount() e get(*idx*) per scorrere una sequenza di DataEvent de
- Finalmente, de offre
  - getDataItem() che restituisce il DataItem modificato
  - getType() che descrive il tipo di modifica (update o cancellazione)

DataAPI: ogni cosa al suo posto!

- Tipicamente, si fa tutto nella Activity:
  - onCreate() → costruzione di gapi
  - onResume() → gapi.connect()
  - onConnected() → addListener(gapi, this)
  - onPause() → removeListener(gapi,this)
gapi.disconnect()
  - onDataChanged() → lettura DataItem aggiornati
- La scrittura di DataItem invece dipende strettamente dalla logica dell'app

DataAPI: esempio

```
public class TestDataAPI extends Activity implements
        DataApi.DataListener,
        GoogleApiClient.ConnectionCallbacks,
        GoogleApiClient.OnConnectionFailedListener {

    private static final String K_NOME = "nome";
    private static final String K_ETA = "eta";
    private GoogleApiClient gapi;

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

        gapi = new GoogleApiClient.Builder(this)
                .addApi(Wearable.API)
                .addConnectionCallbacks(this)
                .addOnConnectionFailedListener(this)
                .build();
    }
```

DataAPI: esempio

```
protected void onResume() {
    super.onStart();
    gapi.connect();
}

public void onConnected(Bundle hint) {
    Wearable.DataApi.addListener(gapi, this);
}

protected void onPause() {
    super.onPause();
    Wearable.DataApi.removeListener(gapi, this);
    gapi.disconnect();
}
```

DataAPI: esempio

```
    public void onDataChanged(DataEventBuffer de) {
        for (DataEvent e : de) {
            if (e.getType() == DataEvent.TYPE_CHANGED) {
                // DataItem aggiornato
                DataItem di = e.getDataItem();
                if (di.getUri().getPath().compareTo("/nonna") == 0) {
                    DataMap dm = DataMapItem.fromDataItem(di).getDataMap();
                    nome = dm.getString(K_NOME);
                    età = dm.getInt(K_ETA);
                    // qui possiamo aggiornare la nonna
                }
            } else if (e.getType() == DataEvent.TYPE_DELETED) {
                // la nonna non c'è più...
            }
        }
    }

    // onConnectionSuspended(), onConnectionFailed() come già visto
}
```

DataAPI: notifiche via Service

- In alternativa al sistema dei callback, possiamo estendere **WearableListenerService**
- Si tratta di un servizio di tipo bound
  - A cui si collega sia la vostra applicazione, sia il processo che esegue GooglePlayServices
- Il Service espone vari metodi che potete (dovete) sovrascrivere nella sottoclasse
  - Per il nostro caso:
    onDataChanged(DataEventBuffer de)

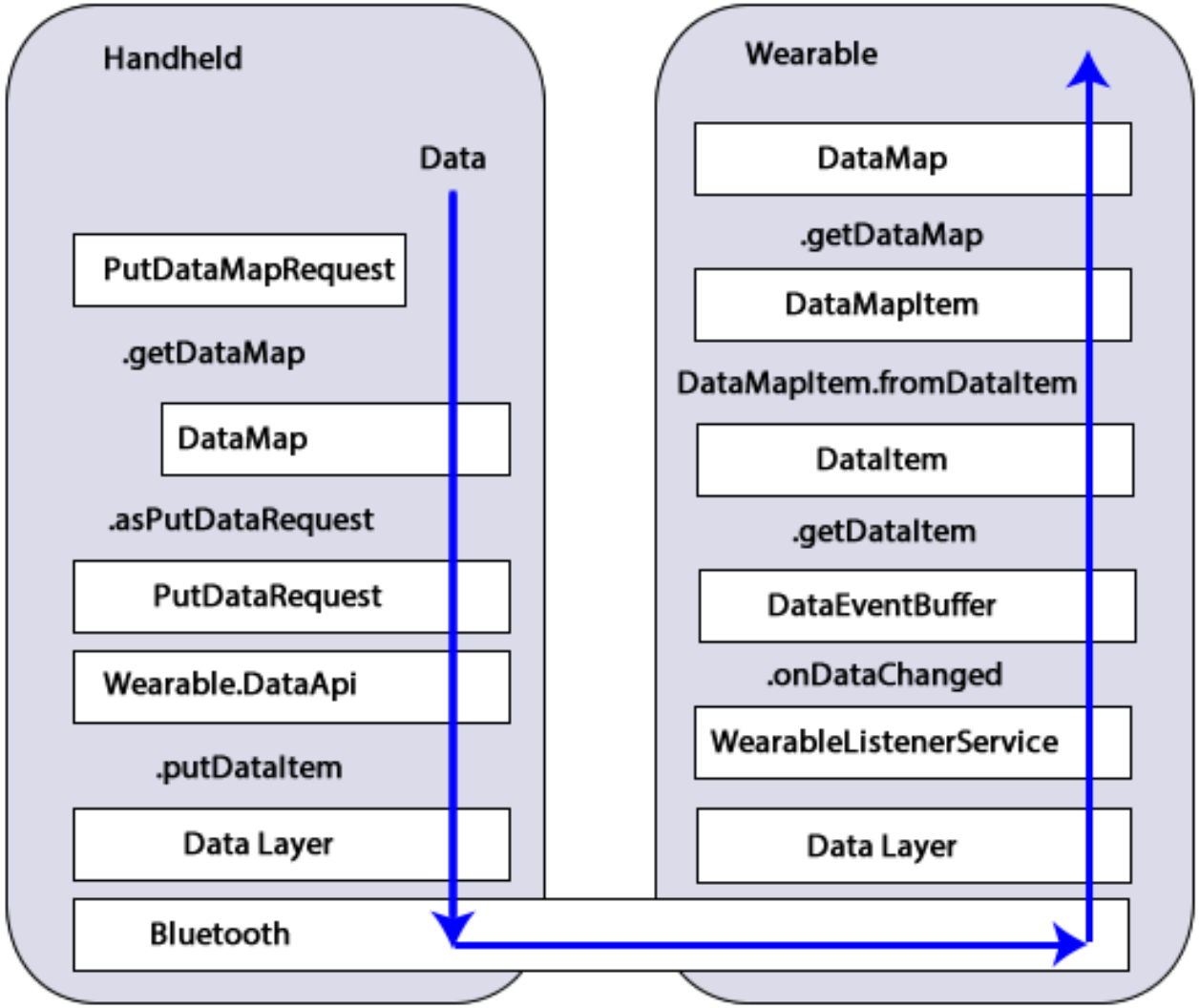
DataAPI: notifiche via Service

- Naturalmente, il Service andrà dichiarato in AndroidManifest.xml
- Deve dichiarare un intent filter che consente a GooglePlayServices di “trovarlo”

```
<service android:name="IlMioWLS">
  <intent-filter>
      <action android:name="com.google.android.gms.wearable.BIND_LISTENER" />
  </intent-filter>
</service>
```

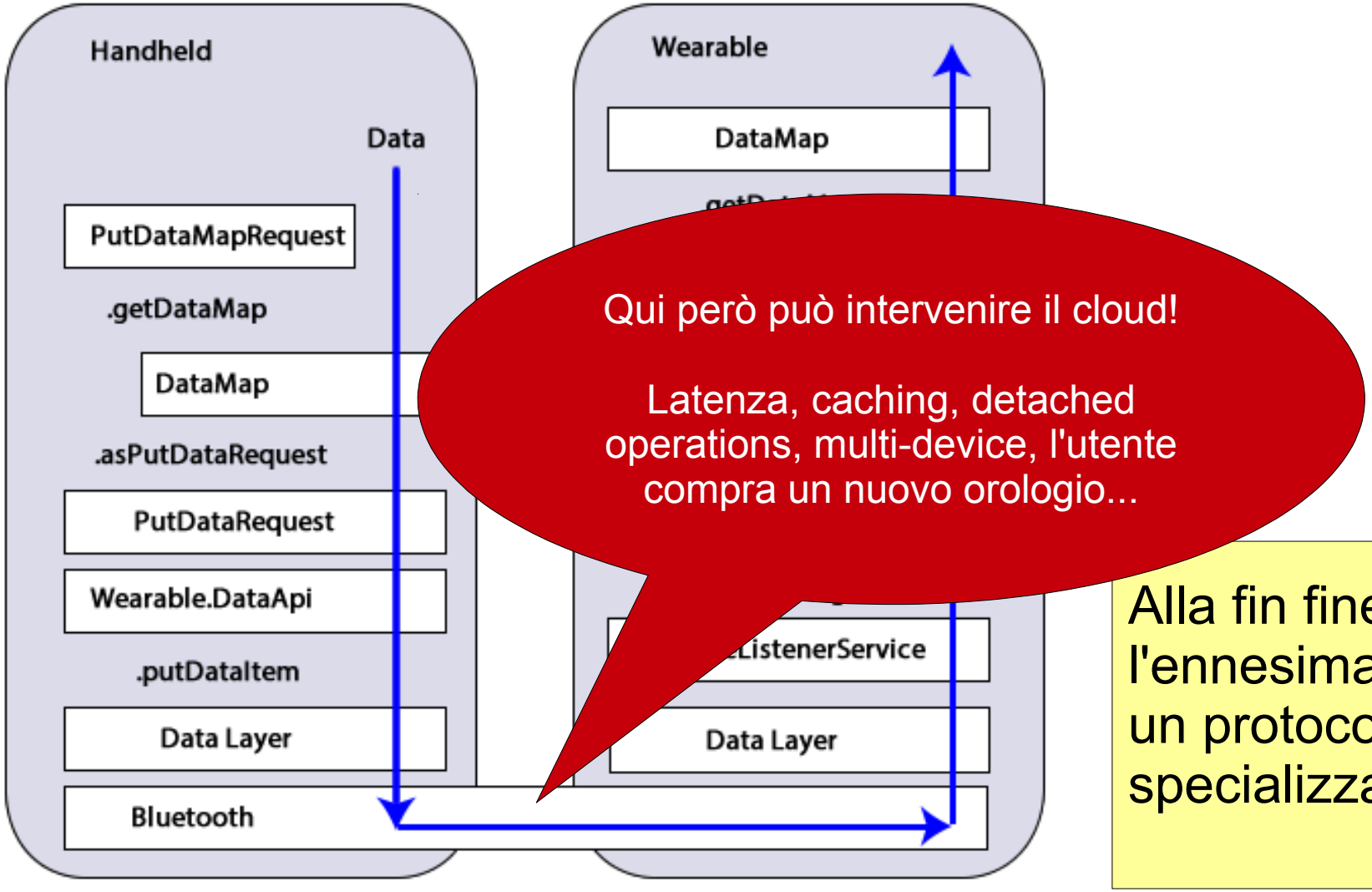
- Nota bene: il ciclo di vita del Service è pur sempre gestito dal sistema
  - Sarà vivo quando ci sono eventi da segnalare, ma Android potrebbe chiuderlo (se non bindato) negli intervalli

DataAPI riassunto



Alla fin fine, è l'ennesima versione di un protocollo RPC, specializzato per i dati

DataAPI riassunto



Qui però può intervenire il cloud!

Latenza, caching, detached operations, multi-device, l'utente compra un nuovo orologio...

Alla fin fine, è l'ennesima versione di un protocollo RPC, specializzato per i dati