HPI Hasso
Plattner
Institut

IT Systems Engineering
Universität Potsdam

Bachelor's Thesis

Accessing and transferring sensor data on Wearables in real-time

Zugriff und Übertragung von Sensor Daten auf Wearables in Echtzeit

by

Stephan Schultz

Potsdam, July 2016

Supervisor

Prof. Dr. Christoph Meinel,
Philipp Berger, Patrick Hennig

Internet-Technologies and Systems Group

Disclaimer

I certify that the material contained in this dissertation is my own work and does not contain significant portions of unreferenced or unacknowledged material. I also warrant that the above statement applies to the implementation of the project and all associated documentation.

Hiermit versichere ich, dass diese Arbeit selbständig verfasst wurde und dass keine anderen Quellen und Hilfsmittel als die angegebenen benutzt wurden. Diese Aussage trifft auch für alle Implementierungen und Dokumentationen im Rahmen dieses Projektes zu.

Potsdam, July 27, 2016

_____

(Stephan Schultz)

Kurzfassung

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Ut purus elit, vestibulum ut, placerat ac, adipiscing vitae, felis. Curabitur dictum gravida mauris. Nam arcu libero, nonummy eget, consectetuer id, vulputate a, magna. Donec vehicula augue eu neque. Pellentesque habitant morbi tristique senectus et netus et malesuada fames ac turpis egestas. Mauris ut leo. Cras viverra metus rhoncus sem. Nulla et lectus vestibulum urna fringilla ultrices. Phasellus eu tellus sit amet tortor gravida placerat. Integer sapien est, iaculis in, pretium quis, viverra ac, nunc. Praesent eget sem vel leo ultrices bibendum. Aenean faucibus. Morbi dolor nulla, malesuada eu, pulvinar at, mollis ac, nulla. Curabitur auctor semper nulla. Donec varius orci eget risus. Duis nibh mi, congue eu, accumsan eleifend, sagittis quis, diam. Duis eget orci sit amet orci dignissim rutrum.

An academic abstract typically outlines four elements relevant to the completed work:

- The research focus (i.e. statement of the problem(s)/research issue(s) addressed)
- The research methods used (experimental research, case studies, questionnaires, etc.)
- The results/findings of the research
- The main conclusions and recommendations

Transate abstract

Abstract

Wearable devices such as smartwatches or activity trackers with embedded sensors are capable of exchanging data with other connected devices. This data will often be transferred to the manufacturer or processed directly on a connected smartphone in order to provide user feedback based on the analyzed data. Almost every wearable device offers third-party developers a way to gain (at least partial) access to the gathered sensor data, allowing custom applications to process them.

In the course of this work, the availability of APIs[1] on different wearables and how likely they can be used to transfer and process sensor data in real-time has been evaluated, as well as the suitability of current devices running proprietary operating systems created by Apple, Google, Jawbone and Microsoft. In addition, an in-depth look at possible implementations for real-time processing of sensor data from devices running Android Wear is included.

---

[1]Application Program Interfaces, provided by the manufacturer

Contents

1. Introduction

When speaking about mobile devices and wearables, it might not be exactly clear which kind of hardware we are talking about.

Mobile devices are smartphones or tablets in this context. Most of the currently available mobile devices offer capabilities that could only be expected from desktop PCs a few years ago.

Wearables can be described as technology gadgets that you can wear on your body, e.g. around your wrist. Usually they are loaded with sensors and can be connected to an other mobile device via Bluetooth. The most commonly used wearables are smartwatches and fitness trackers, but the technology can also live in clothing or jewellery.

1.1. Motivation

The data produced by sensors on wearables can be valuable for multiple use-cases, for example:

- **Activity feedback**: A big selling point for wearables is the ability to track fitness activities in order to improve the health of the person wearing them. Of course this tracking works by analysing sensor data, e.g. to count the steps or track the heart rate.
- **Event triggers**: Many wearables lack large user interfaces - in order to interact with them, one can use gestures. A smartwatch can turn on its screen if you raise your wrist, but in the background a gesture detection system requires access to the device orientation and acceleration.

While some wearables can work as stand-alone devices, they usually depend on connected mobile devices to some extend. That is because the available space for hardware components is very limited, leading to a lower battery and CPU[2] power. However, the device sensors on wearables can produce a large amount of data every second. This data may needs to be pre-processed, analysed or

---

[2]Central processing unit

persisted - all of which requires a lot of processing power and will drain the device battery. Thus, heavy work loads like these should be done on a connected mobile device and require a performant way to transfer data.

1.2. Project Scope

The goal of our Bachelor project "Passwords are Obsolete - Seamless authentication using wearables and mobile devices." was to authenticate a user by utilizing only data from devices that the user already owns. We opted for training classifiers to analyse how a user performs certain activities and to distinguish between behaviour patterns. These classifiers take raw sensor data or extracted features and perform different machine learning methods.

Our unsupervised learning approach required that a wearable can exchange data with a mobile device in a performant jet battery friendly way while keeping the delay in an acceptable range. This requirement is the topic of this thesis, we will implement and evaluate our solution in the upcoming sections.

1.3. Structure

We will start with **Related Work** (2), where we briefly mention similar projects and papers related to this topic. After that, we will let you know why we decided to develop for the Android platform in the **Devices** section (3). In **Concept** (4), we will explain how the basic app setup on mobile and wearable devices needs to look like. Actual code samples will be part of the **Implementation** section (5), where we provide detailed examples for every required functionality. Finally, **Evaluation** (6) will verify our solution using benchmarks and comparisons.

2. Related Work

2.1. Project Abacus

- What's different

2.2. Papers

- Combine abstracts

3. Devices

We had to build prototypes for research purposes and needed to decide which wearables we intend to work with. Although we wanted to support the widest range of devices that we could, we had to ditch some in order to be able to iterate fast. In the following section, we will list the advantages and disadvantages of different wearables.

3.1. Apple Watch

While the currently available Apple Watches all provide sufficient hardware and enough sensors, the software doesn't allow 3rd party developers to take full advantage of this. With WatchOS 2, Apple restricted apps running on the watch to only get access to sensor data while it's visible to the user. For our project however, we needed a way to access sensor data from a background service, which simply isn't possible with the existing APIs. Apple announced that in WatchOS 3 (which isn't available yet), this restriction will be eliminated.

3.2. UP by Jawbone

- 2h API delay
- Can't provide required data frequency

3.3. Microsoft Band

- Awesome, but less users than competition
- Limited to SDK functionality

3.4. Android Wear

Android Wear is a platform for smartwatches that many devices from different manufacturers build upon. Although it's customized to match the conditions of a watch, it's still a full Android OS without any limitations. Because of Androids

open nature, it's possible to use everything that the devices offer without any software restrictions.

Unlike the Apple Watch, Android Wear devices are able to connect to devices that don't belong to the same ecosystem, which increases the number of potential users. Although Apple topped Androids market share by almost 30%[3] in 2015, we decided to develop for the Android platform because of the restrictions mentioned above.

---

[3]Source: IDC Worldwide Quarterly Wearable Device Tracker

4. Concept

As mentioned in section 1, data produced by sensors on wearables needs to be transferred to mobile devices in order to be processed. All Android Wear devices are connected via Bluetooth, we will use this connection to exhange the data with the mobile device.
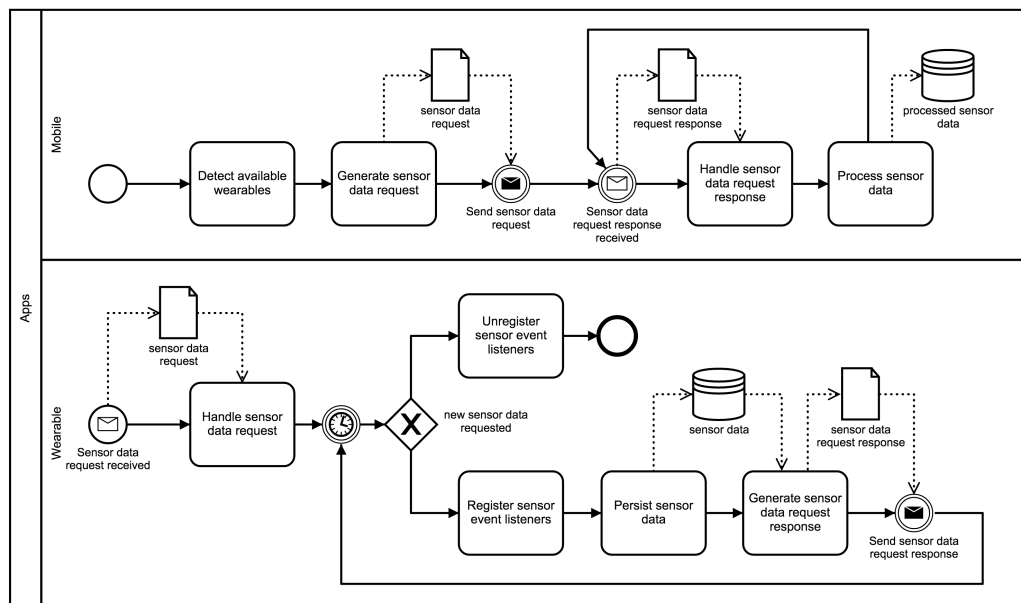

Figure 1: App process model

In order to fulfill the requirements mentioned in section 1.2, we need two different apps to be deployed. One on the mobile device, another one on each wearable device. Figure 1 shows what both apps need to be capable of.

4.1. Mobile App

The app deployed on the mobile device is responsible of:

- Sending sensor data requests
- Handling sensor data request responses
- Processing received sensor data

Once started, it sends a sensor data request to connected wearable devices, which contains information about which sensor data it wants to receive and at which interval (see section 5.4.4). The app then listens for incoming sensor data request responses, which contain the actual sensor data (see section 5.4.5). This data will be processed depending on the usecase.

4.2. Wearable App

The wearable app on the other side takes care of:

- Handling sensor data requests
- Accessing and persisting sensor data
- Sending sensor data request responses

After deployment, it waits for incoming sensor data requests. It registers the required sensor listeners and starts monitoring data changes (see section 5.1.2). Updated sensor data request responses will be periodically sent to the mobile app until the sensor data request changes.

5. Implementation

To showcase and benchmark our work, we created an Android app that visualizes sensor data from the device it runs on and also from connected Android Wear devices. The app is called Sensor Data Logger (see figure 2) and can be downloaded for free from the Google Play Store[4].

Figure 2: Sensor Data Logger App

Code samples in the following sections are snippets from this project and can be seen in context in our GitHub repository[5].

---

[4]https://play.google.com/store/apps/details?id=net.steppschuh.sensordatalogger
[5]https://github.com/Steppschuh/Sensor-Data-Logger

5.1. Accessing Data

Android provides the `SensorManager`[1] system service class in order to grant applications access to the device sensors. The supported sensors can be divided into three categories:

- **Environmantal sensors** (thermometers, barometers and photometers)
- **Motion sensors** (accelerometers, gyroscopes and gravity sensors)
- **Position sensors** (magnetometers and orientation sensors)

Not all sensors are hardware components, the so called "virtual-" or "synthetic sensors" derive their data from one or more hardware-based sensors. Examples for these virtual sensors are the linear acceleration sensor, which computes its data based on the accelerometer and the gravity sensor.

All sensors can be accessed through the Android sensor framework, which provides classes and interfaces that can be used to figure out which sensors are available on the current device, which capabilities they have and what data they produce.

5.1.1. Checking Availability

While most devices have an accelerometer and a magnetometer, only a few have a thermometer. The availability of sensors can't be guaranteed, it's good practice to check this at runtime:

```
1  // get a SensorManager instance
2  SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
3
4  // get a list of available sensors
5  List<Sensor> deviceSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);
6
7  // check if an accelerometer is available
8  Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
9  if (accelerometer != null) {
10    // use accelerometer
11 } else {
12    // perform error handling
13 }
```

If a sensor is available, public methods from the `Sensor`[2] class can be used to get detailed information about it. The name, vendor, version, data range and reporting delay are useful properties, especially because one device can have multiple sensors of the same type.

5.1.2. Monitoring Data Changes

In order to get access to the actual data, a `SensorEventListener`[3] needs to be registered at the `SensorManager` instance. The `SensorEventListener` is an interface which exposes two callback methods:

```java
// create a new SensorEventListener
SensorEventListener listener = new SensorEventListener() {
  @Override
  public void onSensorChanged(SensorEvent event) {
    // sensor reported new data
  }

  @Override
  public void onAccuracyChanged(Sensor sensor, int accuracy) {
    // sensor accuracy changed
  }
};

// specify a reporting delay for the sensor
int delay = SensorManager.SENSOR_DELAY_NORMAL;

// register the listener for the accelerometer
sensorManager.registerListener(listener, accelerometer, delay);
```

The `onSensorChanged` method will be called every time the sensor updates its values. The passed `SensorEvent`[4] holds the sensor, a timestamp, the accuracy and an array of floats containing the actual values.

If the sensor accuracy changes, which often happens when using location sensors, the `onAccuracyChanged` method will be called. It can be usefull to care about these accuracies, the location optained from the Cell-ID or Wi-Fi might be more accurate than the latest GPS coordinates for example. In other cases sensors might need a few seconds to calibrate, like the magnetometer.

When registering a `SensorEventListener`, a delay in microseconds is also passed to the `SensorManager`. It's worth to notice that this value is more like a suggestion, as other applications and the system can alter it. Because the reporting delay can impact the battery life of the device, some manufacturers will lower the reporting interval when the device is idle or the display is turned off.

5.1.3. Monitoring Lifecycle Changes

Once a `SensorEventListener` is registered, the system will keep the requested sensor active and continue to report data, even if the user leaves the application. Hence, one should always unregister listeners as soon as possible in order to prevent battery drain:

```java
public class SensorActivity extends Activity implements SensorEventListener {
  private SensorManager sensorManager;
  private Sensor accelerometer;

  @Override
  public final void onCreate(Bundle savedInstanceState) {
    super.onCreate(savedInstanceState);
    setContentView(R.layout.main);
    sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);
    accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
  }

  @Override
  protected void onResume() {
    super.onResume();
    sensorManager.registerListener(this, accelerometer, SensorManager.SENSOR_DELAY_NORMAL);
  }

  @Override
  protected void onPause() {
    super.onPause();
    sensorManager.unregisterListener(this);
  }

  @Override
  public final void onSensorChanged(SensorEvent event) {
    StringBuilder log = new StringBuilder("Acceleration:");
    log.append(" X: ").append(String.valueOf(event.values[0]));
    log.append(" Y: ").append(String.valueOf(event.values[1]));
    log.append(" Z: ").append(String.valueOf(event.values[2]));
```

```
31      System.out.println(log.toString());
32    }
33
34    @Override
35    public final void onAccuracyChanged(Sensor sensor, int accuracy) {
36      // sensor accuracy changed
37    }
38  }
```

This very basic example activity would print accelerometer data to the console once deployed. Keep in mind that it lacks exception handling, as mentioned in 5.1.1. We presume a basic understanding of how the Android activity[5] lifecycleworks.

5.2. Persisting Data

Once a sensor is reporting data, a new `SensorEvent` will be passed to the callback every few milliseconds. The `values` float array will contain new data, however the system won't allocate a new object for every update in order to improve performance. Handling these values without knowing about its object identity might cause confusion, as they will be overwritten with every update. To prevent this, a new float array can be used to hold the event data:

```
1  @Override
2  public void onSensorChanged(SensorEvent event) {
3    // create a new float array with the same size
4    float[] values = new float[event.values.length];
5
6    // copy data from event values
7    System.arraycopy(event.values, 0, values, 0, event.values.length);
8
9    // persist values in some way
10   persistValues(values);
11 }
```

For our project, we needed to look back at sensor events from the past few seconds in order to detect patterns and to extract features. We created some helper

classes[6] that allowed us to wrap sensor event data in POJOs[6], this way they could be persisted in a batch-like structure.

There is a `Data`[7] class which can wrap the values of a `SensorEvent`, its source and a timestamp. This is necessary because the `SensorEvent` holds references to objects that aren't required multiple times and because there's no public constructor available.

The `DataBatch`[8] class holds and manages a list of `Data` objects. It has a customizable capacity, one can add or remove `Data` and it will automatically remove old `Data` if the capacity has been reached. It also provides some convenience methods, for example to get `Data` from within a given time range. The following code would fill up a `DataBatch` with event data for each requested sensor type:

```java
private Map<Integer, DataBatch> sensorDataBatches = new HashMap<>();

@Override
public void onSensorChanged(SensorEvent event) {
    float[] values = new float[event.values.length];
    System.arraycopy(event.values, 0, values, 0, event.values.length);

    // create a new Data object
    Data data = new Data(values);

    // get a previously initialized DataBatch
    DataBatch dataBatch = getDataBatch(event.sensor.getType());

    // add the new data
    dataBatch.addData(data);
}

public DataBatch getDataBatch(int sensorType) {
    DataBatch dataBatch = sensorDataBatches.get(sensorType);
    if (dataBatch == null) {
        dataBatch = new DataBatch(sensorType);
        sensorDataBatches.put(sensorType, dataBatch);
    }
    return dataBatch;
}
```

---

[6]Plain Old Java Objects

5.3. Serializing Data

At some point, we have to convert the persisted data into byte arrays. We need to serialize objects in order to transfer it to another device or to write it into a file.

The most straightforward solution is using JSON[7], which is a common data-interchange format. It is easy to read for humans and easy to parse for software, which is why we decided to use this format. Fortunately, POJOs can be directly mapped to JSON name-value pairs. Existing libraries like gson[8] or jackson[9] are very well known and provide interfaces that make JSON handling very uncomplicated.

The `DataRequestResponse`[9] class for example holds a list of `DataBatches` and is responsible for exchaning sensor data with the connected mobile device. For convenience, all classes that we transfer implement methods that can be used for JSON serialization and deserialization.

```java
@JsonIgnore
public String toJson() {
  String jsonData = null;
  try {
    ObjectMapper mapper = new ObjectMapper();
    mapper.enable(SerializationFeature.INDENT_OUTPUT);
    mapper.disable(SerializationFeature.FAIL_ON_EMPTY_BEANS);
    jsonData = mapper.writeValueAsString(this);
  } catch (Exception ex) {
    ex.printStackTrace();
  }
  return jsonData;
}

public static DataRequestResponse fromJson(String json) {
  DataRequestResponse dataRequestResponse = null;
  try {
    ObjectMapper mapper = new ObjectMapper();
    mapper.disable(SerializationFeature.FAIL_ON_EMPTY_BEANS);
    dataRequestResponse = mapper.readValue(json, DataRequestResponse.class);
  } catch (Exception ex) {
```

---

[7]JavaScript Object Notation
[8]https://github.com/google/gson
[9]https://github.com/FasterXML/jackson

```
22      ex.printStackTrace();
23    }
24    return dataRequestResponse;
25  }
```

This code block is part of the `DataRequestResponse` class. The `toJson()` method writes the current object state into a JSON string, while `fromJson (String json)` creates a new `DataRequestResponse` object from a given JSON string. The JSON string can be converted to a byte array, which can be transferred as described in section 5.4. It is crucial that the same character encoding is used, we stick with UTF-8.

5.4. Transferring Data

Actually using the sensor event data requires transferring it to a device with sufficient processing power.

5.4.1. General Approach

In a world where the IoT[10] is a big topic, data is usually uploaded to the cloud and processed on powerful servers. Although one could do that from a mobile device, this approach would produce a huge amount of traffic and ultimately lead to privacy concerns.
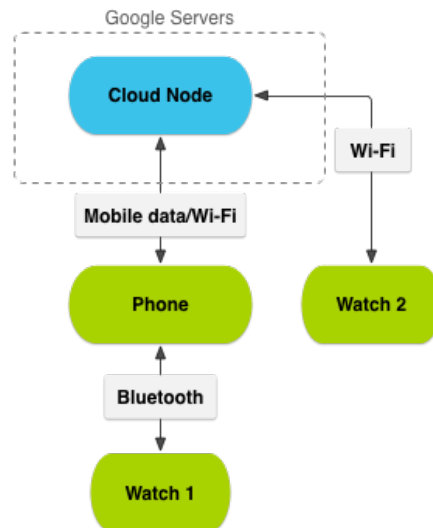
---

[10]Internet of Things

Figure 3: Sample network with mobile and wearable nodes[11]

The setup that this work is about could be seen as a peer to peer network between a mobile device and multiple wearables. Because the devices are connected with each other, there's no need to detour data through the internet. By default, wearables are connected via Bluetooth with a mobile device. On the Android platform, this connection can be accessed through the `WearableApi`[10].

---

[11]https://developer.android.com/training/wearables/data-layer/

5.4.2. The Wearable Data Layer

The Data Layer API is part of of the Google Play Services. It contains the only APIs that should be used to set up a communication channel between a mobile device and wearable devices. Custom, low-level socket implementations are not recommended.

As shown in figure 3, the Data Layer API can handle multiple nodes at once. We are particularly interested in the Bluetooth channel, but we don't have to care about the connection because this is handled by the Play Services for us. The API can be reached using an instance of the `GoogleApiClient`[11]. Because some Play Services may not be available on every device, the `GoogleApiClient` needs to be setup first:

```
1  GoogleApiClient googleApiClient = new GoogleApiClient.Builder(context)
2      .addConnectionCallbacks(new ConnectionCallbacks() {
3          @Override
4          public void onConnected(Bundle connectionHint) {
5              // start using the Data Layer API
6          }
7          @Override
8          public void onConnectionSuspended(int cause) {
9              // something interrupted the connection
10          }
11      })
12      .addOnConnectionFailedListener(new OnConnectionFailedListener() {
13          @Override
14          public void onConnectionFailed(ConnectionResult result) {
15              // API might not be available
16          }
17      })
18      .addApi(Wearable.API)
19      .build();
```

This code block would initialize a `GoogleApiClient` instance. However, `addApi(Wearable.API)` would cause a call to `onConnectionFailed` if the device it runs on doesn't have the Android Wear app[12] installed. This app is required because it handles the connection and synchronization of wearables. For more gracefull error handling, `addApiIfAvailable(Wearable.API)` might be the

---

[12]https://play.google.com/store/apps/details?id=com.google.android.wearable.app

more appropriate solution.

5.4.3. The Message API

There are multiple ways of exchanging data between nodes using the Data Layer API. While the `DataApi`[12] can be used to synchronize larger binary blobs (`Assets`[13]) accross the wearable network, the `MessageApi`[14] is more suitable for exchanging smaller amounts of data. A message consists of the following items:

- **Path**: A string that uniquely identifies the message action.
- **Payload**: An optional byte array.

Payloads are not required by default because messages are a one-way communication mechanism, often used to only trigger RPCs[13]. We use this to pass a serialized `DataRequest`[15] or `DataRequestResponse`[9].

5.4.4. Sending Messages

In order to send a message to a connected wearable, the `Node`[16] representation of that device is required. We can use the `NodeApi`[17] to query connected nodes:

```
1  // a list that holds available nodes
2  private List<Node> nearbyNodes;
3
4  private void updateNearbyNodes() {
5    Wearable.NodeApi.getConnectedNodes(googleApiClient)
6        .setResultCallback(new ResultCallback<NodeApi.GetConnectedNodesResult>() {
7          @Override
8          public void onResult(NodeApi.GetConnectedNodesResult nodes) {
9            nearbyNodes = new ArrayList<Node>();
10
11           // add all nearby nodes to the list
12           for (Node node: nodes.getNodes()) {
13             if (node.isNearby()) {
14               nearbyNodes.add(node);
15             }
```

---
[13]Remote procedure calls

```
16            }
17          }
18        });
19  }
```

The `nearbyNodes` list now contains all currently connected wearables. We can get a display name and an id for each node, which we need to select our message target. For simplicity, the following code block sends a message to all nearby nodes:

```
1   private void startRequestingSensorData() {
2     // send a request message to all nodes
3     sendMessageToNearbyNodes("/start_requesting_sensor_data", null);
4   }
5
6   private void sendMessageToNearbyNodes(String path, byte[] payload) {
7     for (Node node: nearbyNodes) {
8       sendMessageToNode(node.getId(), path, payload);
9     }
10  }
11
12  private void sendMessageToNode(String nodeId, String path, byte[] payload) {
13    Wearable.MessageApi.sendMessage(googleApiClient, nodeId, path, payload)
14        .setResultCallback(new ResultCallback() {
15          @Override
16          public void onResult(SendMessageResult sendMessageResult) {
17            if (!sendMessageResult.getStatus().isSuccess()) {
18              // perform exception handling
19            }
20          }
21        });
22  }
```

Note that we can pass `null` as a payload if no data is required. Instead of null, a serialized `DataRequest` object could be passed. The receiving nodes could deserialize it to find out which sensors are requested or at which interval they should report updates.

19

5.4.5. Receiving Messages

Apps running on the wearables need to implement the `MessageListener`[18] interface in order to get notified about incoming messages. These listeners have to be registered using the `MessageApi.addListener()` function.

```java
@Override
public void onMessageReceived(MessageEvent messageEvent) {
    if (messageEvent.getPath().equals("/start_requesting_sensor_data")) {
        // get the message payload
        byte[] payload = messageEvent.getData();

        // process the request
        startTransferringSensorData();
    }
}
```

Obviously, message paths should be static and final constants defined in a shared module that the mobile and wearable app package have access to. In our implementation, the payload would be a serialized `DataRequest`.

Explain ChannelApi

5.5. Putting it all together

asdf

Change section title

20

6. Evaluation

6.1. Benchmarks

In order to measure how performant different implementations are, we created different benchmarks. These helped us to evaluate which parts of our solution require optimization. Each benchmark contains the average result of 500 consequtive measurements and has been validated multiple times.

6.1.1. Setup

For the benchmarks, we created a `TimeTracker`[19] that is capable of measuring the delay in nanoseconds between events. It also provides convenience functions for merging repetitive measurements to avoid round-off errors.

We measured on our test devices, a Nexus 9 tablet (released November 2014, SDK 24) and a LG G Watch R (released October 2014, SDK 23). Both ran the latest Android version and are a good representation for currently available high-end devices. The devices were placed next to each other on a table, with quite a lot of other electronic devices nearby. We also altered the distance between the devices but figured out that it had no significant impact on the measurements.

6.1.2. Data Transmission Delay

The most important performance indicator is the time it takes between these events: *A sensor has updated values* (on the wearable device) and *Updated sensor values have been received* (on the mobile device). For this benchmark, the measured operations include:

- Creating `DataBatches`[8] with the latest sensor data (wearable)
- Serializing a `DataRequestResponse`[9] containing that data (wearable)
- Transferring the data using the `MessageApi`[14] (wearable & mobile)
- Deserializing the `DataRequestResponse` (mobile)

In order to reduce the serialization overhead, we collect sensor data in `DataBatches`

[8] before transferring it to a mobile device. This approach drastically improves the *delay per sent byte* ratio, because we have to serialize and deserialize less messages and less meta data. However, it also delays the data depending on the batch capacity. Just like buffering a stream, this is a trade-off between beeing less efficient or beeing less real-time.

For the measurements in table 1, we batched sensor data from the accelerometer for **50 milliseconds** before transferring it while altering the sensor delay:

delay in ns	bytes	delay / bytes	comment
1,266,250,000	200	~6,331,250	~1 update (`SENSOR_DELAY_NORMAL`)
1,271,160,000	482	~2,637,261	~2 updates (`SENSOR_DELAY_UI`)
1,285,510,000	1,056	~1,217,339	~6 updates (`SENSOR_DELAY_GAME`)
1,306,400,000	3,599	~362,989	~24 updates (`SENSOR_DELAY_FASTEST`)

Table 1: Transmission delay, 50ms batches

When using `SENSOR_DELAY_NORMAL`, the sensor reports only about one update during the batching duration. This basically results in no improvement at all because we still have to deal with the serialization overhead for each update. By collecting more updates in the same time frame (in this case by switching to `SENSOR_DELAY_FASTEST`), we were able to boost the efficency by 94.3% while only raising the delay by 3.2%.

We wanted to improve even further and increased the batching duration to **500 milliseconds**, as measured in table 2:

delay in ns	bytes	delay / bytes	comment
1,329,120,000	625	~2,126,592	~3 updates (`SENSOR_DELAY_NORMAL`)
1,331,970,000	1,340	~994,007	~8 updates (`SENSOR_DELAY_UI`)
1,373,370,000	8,262	~166,227	~28 updates (`SENSOR_DELAY_GAME`)
1,412,810,000	16,951	~83,346	~118 updates (`SENSOR_DELAY_FASTEST`)

Table 2: Transmission delay, 500ms batches

Increasing the duration resulted in more updates per transferred `DataRequestResponse`

[9]. Compared to the first measurement in table 1, we were able to transfer 84 times more bytes at the cost of only 147 milliseconds.

For some use cases, less frequent data might be sufficient. Instead of altering the reporting delay of the sensor, we can also sent data from multiple sensors simultaneously. Table 3 shows how **multiple, 3-dimensional sensors** perform:

delay in ns	bytes	delay / bytes	comment
1,452,400,000	16,690	~87,022	~116 updates, 1 sensor
1,489,100,000	22,819	~65,257	~246 updates, 2 sensors
1,752,420,000	60,679	~28,880	~512 updates, 4 sensors
2,842,640,000	177,495	~16,015	~1504 updates, 8 sensors

Table 3: Transmission delay, multiple sensors

Keeping in mind that this transfer is performed multiple hundreds or thousands times, these efficency improvements sum up and save a lot of computing and battery power on both devices.

6.1.3. Serialization

As mentioned before, serialization and deserialization are responsible for a large part of the processing time. It is crucial that this part is optimized, that's why we opted for utilizing established libraries. Namely, we decided to use jackson[14] because of its performance advantage compared to other well known libraries like gson[15].

There are benchmarks available that compared these JSON libraries, which is why we won't present new measurements here. The key take-away is that jackson is faster in handling larger files, while gson should be used for smaller files[16].

---

[14]https://github.com/FasterXML/jackson
[15]https://github.com/google/gson
[16]http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/

6.1.4. Data Rate

asdf

6.1.5. Battery Impact

asdf

7. Future Work

7.1. Channel API

In our implementation, we utilized the `MessageApi`[14] because it is recommended for smaller messages. We also used it for other project related purposes, so we sticked to it for consistency. However, we figured out that there might be an even more performant solution:

The `ChannelApi`[20] is also part of the `WearableApi`[10], accessible through the `GoogleApiClient`[11]. It should be used to transfer large files, just like the already mentioned `DataApi`[12]. In contrast to the `DataApi`, it doesn't synchronize `Assets`[13] accross the devices. Instead, it creates a bidirectional channel that the sending and receiving node may read or write to. Data can be exchanged using byte streams, which makes the `ChannelApi` very suitable for transferring streamed content like music - or even sensor data.

Unfortunately we weren't able to evaluate whether implementing the `ChannelApi` would increase our performance jet.

7.2. Compression

In order to decrease the amount of data transferred with each `DataRequestResponse` [9], we can try to reduce the overhead created by meta data. This overhead can only be reduced to a certain amount, though.

Each sensor data update is an array of values, and can have up to 9 dimensions (depending on the sensor). Each dimension holds a single-precision 32-bit IEEE 754 floating point. In our use case, we didn't need values with that high precision. Instead, rounded values would be sufficient for our algorithms.

By switching the data type for our data arrays from floats to shorts (16-bit signed two's complement integers), we could save about 50% of transferred bytes. However, it still has to be evaluated whether the loss of precision and the processing power required for converting the data justifies this saving.

7.3. Open Research Questions

7.4. Extensions

8. Conclusion

References

[1] Google Developers: *Sensor manager.*
   https://developer.android.com/reference/android/
      hardware/SensorManager.html.

[2] Google Developers: *Sensor.*
   https://developer.android.com/reference/android/
      hardware/Sensor.html.

[3] Google Developers: *Sensor event listener.*
   https://developer.android.com/reference/android/
      hardware/SensorEventListener.html.

[4] Google Developers: *Sensor event.*
   https://developer.android.com/reference/android/
      hardware/SensorEvent.html.

[5] Google Developers: *Activity.*
   https://developer.android.com/reference/android/app/
      Activity.html.

[6] Stephan Schultz: *Sensor data logger: Data package.*
   https://github.com/Steppschuh/Sensor-Data-Logger/tree/
      master/Android%20Source/datalogger/src/main/java/
      net/steppschuh/datalogger/data.

[7] Stephan Schultz: *Sensor data logger: Data.*
   https://github.com/Steppschuh/Sensor-Data-Logger/blob/
      master/Android%20Source/datalogger/src/main/java/
      net/steppschuh/datalogger/data/Data.java.

[8] Stephan Schultz: *Sensor data logger: Data batch.*
   https://github.com/Steppschuh/Sensor-Data-Logger/blob/
      master/Android%20Source/datalogger/src/main/java/
      net/steppschuh/datalogger/data/DataBatch.java.

[9] Stephan Schultz: *Sensor data logger: Data request response.*
   https://github.com/Steppschuh/Sensor-Data-Logger/
      blob/master/Android%20Source/datalogger/src/
      main/java/net/steppschuh/datalogger/data/request/
      DataRequestResponse.java.

[10] Google Developers: *Wearable api*.
https://developers.google.com/android/reference/com/
google/android/gms/wearable/Wearable.

[11] Google Developers: *Google api client*.
https://developers.google.com/android/reference/com/
google/android/gms/common/api/GoogleApiClient.

[12] Google Developers: *Data api*.
https://developers.google.com/android/reference/com/
google/android/gms/wearable/DataApi.html.

[13] Google Developers: *Asset*.
https://developers.google.com/android/reference/com/
google/android/gms/wearable/Asset.

[14] Google Developers: *Message api*.
https://developers.google.com/android/reference/com/
google/android/gms/wearable/MessageApi.

[15] Stephan Schultz: *Sensor data logger: Data request*.
https://github.com/Steppschuh/Sensor-Data-Logger/blob/
master/Android%20Source/datalogger/src/main/java/
net/steppschuh/datalogger/data/request/DataRequest.
java.

[16] Google Developers: *Node*.
https://developers.google.com/android/reference/com/
google/android/gms/wearable/Node.

[17] Google Developers: *Node api*.
https://developers.google.com/android/reference/com/
google/android/gms/wearable/NodeApi.

[18] Google Developers: *Message listener*.
https://developers.google.com/android/reference/
com/google/android/gms/wearable/MessageApi.
MessageListener.

[19] Stephan Schultz: *Sensor data logger: Time tracker*.
https://github.com/Steppschuh/Sensor-Data-Logger/blob/
master/Android%20Source/datalogger/src/main/java/
net/steppschuh/datalogger/logging/TimeTracker.java.

[20] Google Developers: *Channel api.*
`https://developers.google.com/android/reference/com/`
`google/android/gms/wearable/ChannelApi.`

A. Appendix

Lorem ipsum dolor sit amet, consectetur adipiscing elit. In erat mauris, faucibus quis pharetra sit amet, pretium ac libero. Etiam vehicula eleifend bibendum. Morbi gravida metus ut sapien condimentum sodales mollis augue sodales. Vestibulum quis quam at sem placerat aliquet. Curabitur a felis at sapien ullamcorper fermentum. Mauris molestie arcu et lectus iaculis sit amet eleifend eros posuere. Fusce nec porta orci.

Integer vitae neque odio, a sollicitudin lorem. Aenean orci mauris, tristique luctus fermentum eu, feugiat vel massa. Fusce sem sem, egestas nec vulputate vel, pretium sit amet mi. Fusce ut nisl id risus facilisis euismod. Curabitur et elementum purus. Duis tincidunt fringilla eleifend. Morbi id lorem eu ante adipiscing feugiat. Sed congue erat in enim eleifend dignissim at in nisl. Donec tortor mauris, mollis vel pretium vitae, lacinia nec sapien. Donec erat neque, ullamcorper tincidunt iaculis sit amet, pharetra bibendum ipsum. Nunc mattis risus ac ante consequat nec pulvinar neque molestie. Etiam interdum nunc at metus lacinia non varius erat dignissim. Integer elementum, felis id facilisis vulputate, ipsum tellus venenatis dui, at blandit nibh massa in dolor. Cras a ultricies sapien. Vivamus adipiscing feugiat pharetra.

Notes