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 30, 2016

(Stephan Schultz)

Kurzfassung

Tragbare Geräte, wie Smartwatches oder Fitness Tracker mit integrierten Sensoren, sind in der Lage Daten mit anderen verbundenen Geräten auszutauschen. Diese Daten werden häufig an die Hersteller übertragen oder direkt auf einem verbundenen Smartphone verarbeitet. So kann Nutzern Feedback basierend auf den gemessenen Daten gegeben werden. Nahezu jedes dieser tragbaren Geräte bietet Entwicklern die Möglichkeit, auf die Messwerte zuzugreifen, um diese mit eigener Software zu verarbeiten.

Im Rahmen dieser Arbeit wurde die Verfügbarkeit von APIs[1] auf verschiedenen Geräten und dessen Eignung zur Übertragung von Sensor Daten in Echtzeit evaluiert. Funktionale Implementierungen für die Datenübertragung zwischen tragbaren Geräten und Smartphones basierend auf der Android Plattform wurden präsentiert. Die Leistung, Effizienz und der Akkuverbrauch wurden basierend auf Messungen analysiert.

[1] Application Program Interfaces, Schnittstellen zur Anwendungsprogrammierung

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[2] on different wearables and how likely they can be used to transfer and process sensor data in real-time has been evaluated. We have presented functional implementations for transferring data between wearables and mobile devices powered by the Android platform. Performance, efficiency and battery impact have been analyzed using benchmarks.

[2] 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 jewelery.

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 analyzing 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[3] power. However, the device sensors on wearables can produce a large amount of data every second. This data may needs to be pre-processed, analyzed or

[3] 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 analyze how a user performs certain activities and to distinguish between behavior patterns. These classifiers take raw sensor data or extracted features and perform different machine learning methods.

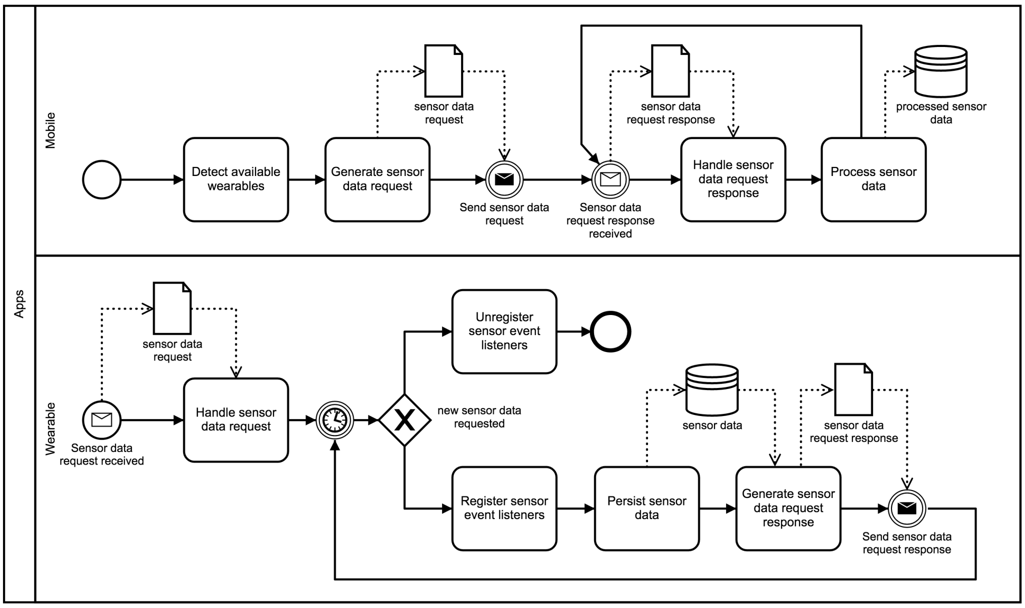

Figure 1: Trust level approach

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, structured as described below:

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. In **Evaluation** (6) we will verify our solution using benchmarks and comparisons. Possible improvements will be described in **Future work** (7). Finally, **Conclusion** (8) will wrap up our work.

2. Related Work

2.1. Motion Leaks through Smartwatch Sensors

Wang et al. evaluated whether it is possible to figure out what a user is typing on a keyboard, just by looking at the sensor data that smartwatches produce. In their paper "MoLe: Motion Leaks through Smartwatch Sensors"[1], they described how they analyzed peoples typing Behavior in order to identify possible information leaks when users type while wearing a smartwatch. They also developed a system for the Samsung Gear Live smartwatch (one of the first devices powered by Android Wear) that resembles motion data to commonly used english words.

Although MoLe turned out to be able to detect typed words with an high accuracy, they used the wearable only to record the sensor data. Files containing the data were exported from the watch and processed on powerful PCs and not on mobile devices, which does not meat our project's requirements.

2.2. Real-Time Sensing on Android

In "Real-Time Sensing on Android"[2], Yin Yan et al. examined Android's sensor architecture and whether it is suitable for use in a real-time context. They took an in-depth look at the very low-level implementations of the Android `SensorManager`[3], including the kernel, HAL[4], and the `SensorService` (which polls raw sensor data through the HAL). Their research showed that Android's sensor architecture does not provide predictable sensing. It does not have any priority support in sensor data delivery, because all sensor data follows a single path from the kernel to apps. Also, the amount of time it takes to deliver sensor data is unpredictable because Android relies heavily on polling and buffering.

For our project however, we worked around these limitations by implementing algorithms that abstracted from the data frequency and delay.

[4]Hardware Abstraction Layer

2.3. Project Abacus

Just like our project, Google wants to get rid of passwords using mobile devices. The Advanced Technology and Projects group developed a product codenamed "Project Abacus", which is constantly paying attention to how a user is interacting with a mobile device. It combines multiple factors, including how a users types, voice and face detection, and how apps are used. The project was recently re-branded as the "Trust API", which provides third-party developers access to a score calculated by the system.

Although we have no doubt that Google is able to perform all required steps without drastically impacting the device's battery life, we can not except that data will be sent to Google servers. This ultimately leads to privacy concerns. In addition to that, there are no plans to support wearable devices or to make the API accessible for services which are not running on a user's mobile device.

2.4. Recognizing ADLs in Real Time

Waldhör Klemens and Rob Baldauf published their work about activity detection in "Recognizing Drinking ADLs in Real Time using Smartwatches and Data Mining"[4]. They developed an app that is capable of detecting ADLs[5] related to drinking. Running on the Samsung Gear S (powered by Tizen OS), the app is collecting sensor data and repeatedly fitting models onto it in order to detect activities. These models have previously been generated by applying data mining (logistic regression, neural networks, discriminant analysis, and random trees) on features extracted from recorded sample data. This way, they achieved a classification accuracy of 92% to 97%, depending on the models used.

The authors' approach is very similar to ours, but they decided to execute all data processing directly on the wearable instead of outsourcing it to a mobile device. Because of that, they ended up having huge issues with energy management. We avoided this by using the solution presented in the course of this work.

[5] Activities Of Daily Living

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. Namely, we evaluated the Apple Watch, the UP and Microsoft Band as well as Android watches.

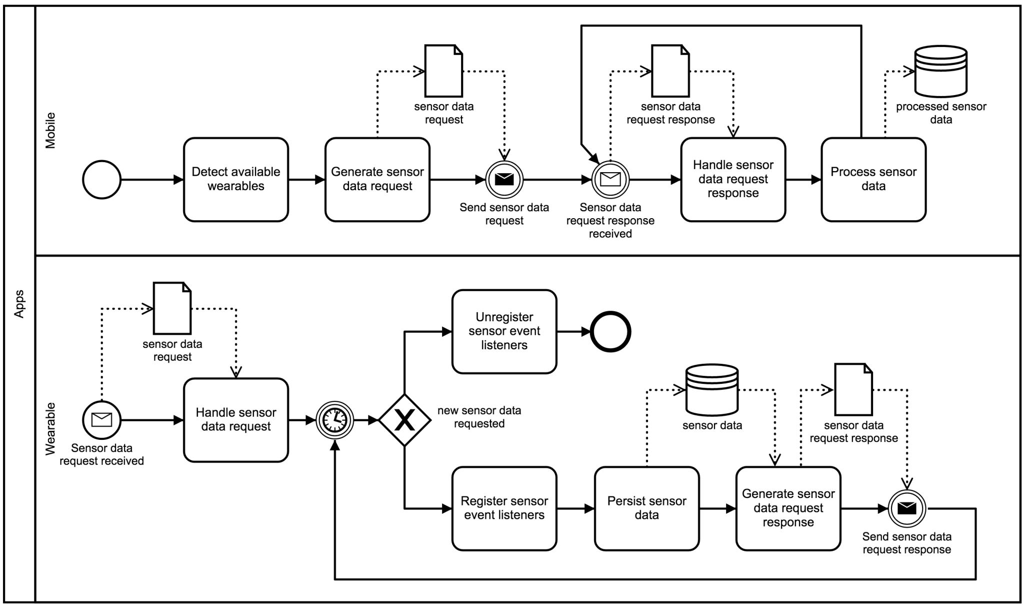

Figure 2: Wearable devices

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.

3.5. Decision

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%[6] in 2015, we decided to develop for the Android platform because of the restrictions mentioned above.

[6]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 exchange the data with the mobile device.

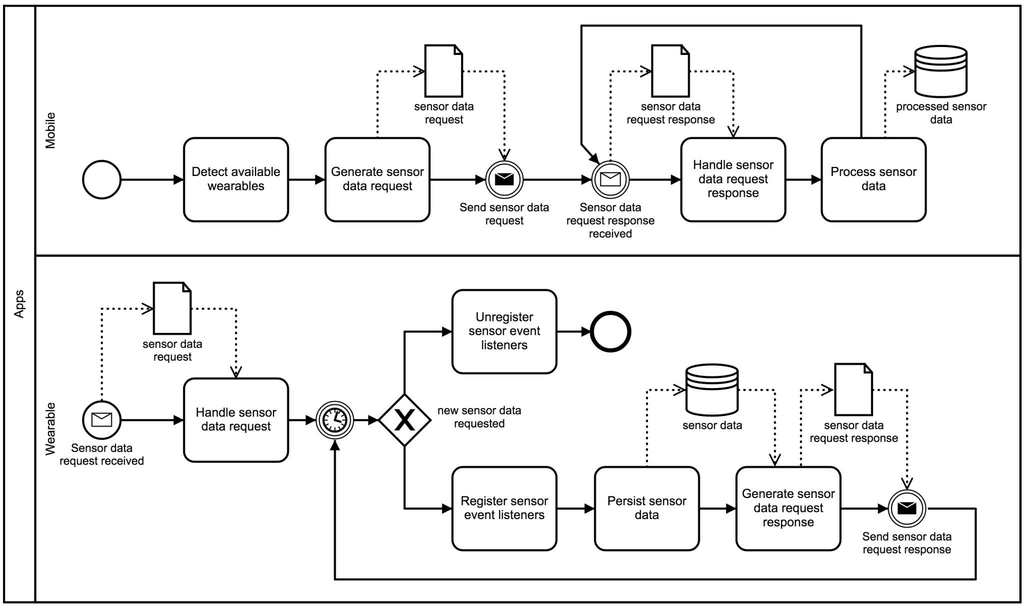

Figure 3: App process model

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 3 shows what both apps need to be capable of.

4.1. Mobile App

The app deployed on the mobile device is responsible for:

- 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 use case.

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.

More detailed app descriptions

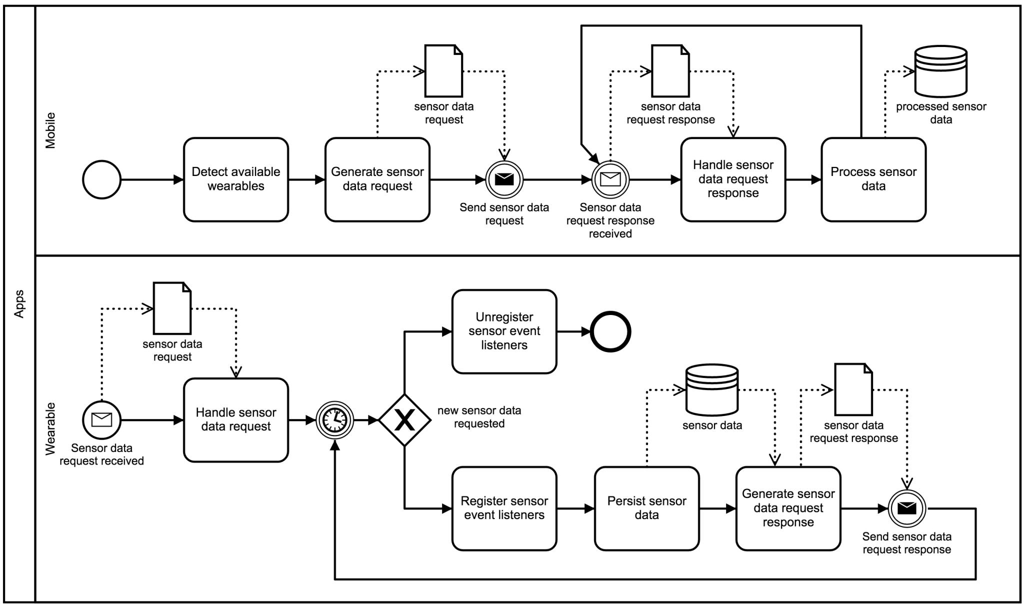

Figure 4: Mobile and Wearable apps

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 5) and can be downloaded for free from the Google Play Store[7].

Figure 5: 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[8].

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

5.1. Accessing Data

Android provides the `SensorManager`[3] system service class in order to grant applications access to the device sensors. The supported sensors can be divided into three categories:

- **Environmental 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:

```
// get a SensorManager instance
SensorManager sensorManager = (SensorManager) getSystemService(Context.SENSOR_SERVICE);

// get a list of available sensors
List<Sensor> deviceSensors = sensorManager.getSensorList(Sensor.TYPE_ALL);

// check if an accelerometer is available
Sensor accelerometer = sensorManager.getDefaultSensor(Sensor.TYPE_ACCELEROMETER);
if (accelerometer != null) {
  // use accelerometer
} else {
  // perform error handling
}
```

If a sensor is available, public methods from the `Sensor`[5] 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`[6] needs to be registered at the `SensorManager` instance. The `SensorEventListener` is an interface which exposes two callback methods:

```
// 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`[7] 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 useful to care about these accuracies, the location obtained 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.

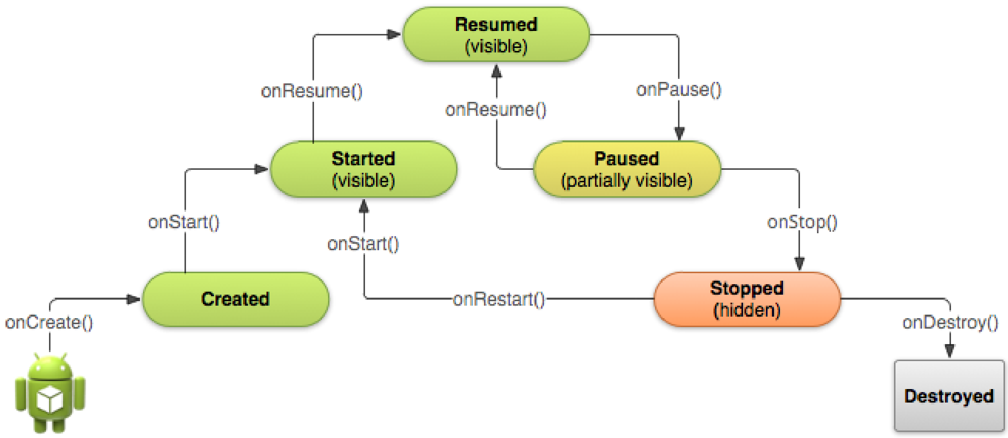

Figure 6: Activity lifecycle callbacks[9]

We need to have elemental understanding of how the Android `Activity`[8] lifecycle works, as illustrated in figure 6. For a basic implementation, we need to override three `Activity` methods:

- `onCreate()` is called when the `Activity` is first created. This is wehere we will create our view and setup our `SensorManager`.

[9]https://developer.android.com/training/basics/activity-lifecycle/starting.html

- `onResume()` is called when the activity is at the top of the activity stack and starts interacting with the user. We will register our `SensorEventListeners` here.
- `onPause()` is called when the system is about to start resuming a previous activity. We will unregister our `SensorEventListeners` here.

The following example `Activity` would print accelerometer data to the console while the app is visible to the user. Keep in mind that it lacks exception handling, as mentioned in 5.1.1:

```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]));
    System.out.println(log.toString());
  }

  @Override
  public final void onAccuracyChanged(Sensor sensor, int accuracy) {
```

```
    // sensor accuracy changed
  }
}
```

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:

```java
@Override
public void onSensorChanged(SensorEvent event) {
  // create a new float array with the same size
  float[] values = new float[event.values.length];

  // copy data from event values
  System.arraycopy(event.values, 0, values, 0, event.values.length);

  // persist values in some way
  persistValues(values);
}
```

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[9] that allowed us to wrap sensor event data in POJOs[10], this way they could be persisted in a batch-like structure.

There is a `Data`[10] 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`[11] class holds and manages a list of `Data` objects. It has a

[10]Plain Old Java Objects

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:

```
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;
}
```

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 them to another device or to write it into a file.

The most straightforward solution is using JSON[11], which is a common data-

[11] JavaScript Object Notation

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[12] or jackson[13] are very well known and provide interfaces that make JSON handling very uncomplicated.

The `DataRequestResponse`[12] class for example holds a list of `DataBatches` and is responsible for exchanging 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.

```
@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) {
    ex.printStackTrace();
  }
  return dataRequestResponse;
}
```

This code block is part of the `DataRequestResponse` class. The `toJson()` method writes the current object state into a JSON string, while `fromJson`

[12]https://github.com/google/gson
[13]https://github.com/FasterXML/jackson

`(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[14] 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.

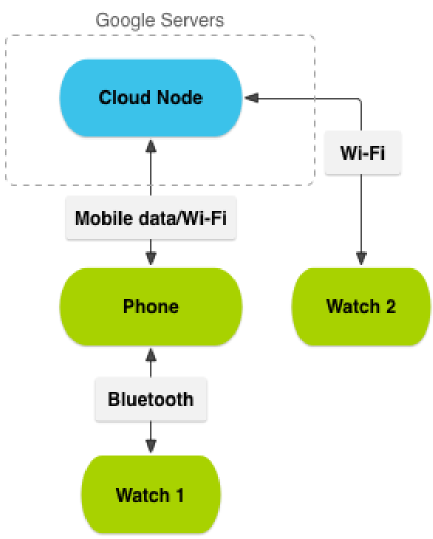

Figure 7: Sample network with mobile and wearable nodes[15]

[14]Internet of Things

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`[13].

[15] 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 7, 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`[14]. Because some Play Services may not be available on every device, the `GoogleApiClient` needs to be setup first:

```
GoogleApiClient googleApiClient = new GoogleApiClient.Builder(context)
    .addConnectionCallbacks(new ConnectionCallbacks() {
      @Override
      public void onConnected(Bundle connectionHint) {
        // start using the Data Layer API
      }
      @Override
      public void onConnectionSuspended(int cause) {
        // something interrupted the connection
      }
    })
    .addOnConnectionFailedListener(new OnConnectionFailedListener() {
      @Override
      public void onConnectionFailed(ConnectionResult result) {
        // API might not be available
      }
    })
    .addApi(Wearable.API)
    .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[16] installed. This app is required because it handles the connection and synchronization of wearables. For more graceful error handling, `addApiIfAvailable(Wearable.API)` might be the

[16] 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`[15] can be used to synchronize larger binary blobs (`Assets`[16]) across the wearable network, the `MessageApi`[17] 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[17]. We use this to pass a serialized `DataRequest`[18] or `DataRequestResponse`[12].

5.4.4. Sending Messages

In order to send a message to a connected wearable, the `Node`[19] representation of that device is required. We can use the `NodeApi`[20] to query connected nodes:

```java
// a list that holds available nodes
private List<Node> nearbyNodes;

private void updateNearbyNodes() {
  Wearable.NodeApi.getConnectedNodes(googleApiClient)
      .setResultCallback(new ResultCallback<NodeApi.GetConnectedNodesResult>() {
        @Override
        public void onResult(NodeApi.GetConnectedNodesResult nodes) {
          nearbyNodes = new ArrayList<Node>();

          // add all nearby nodes to the list
          for (Node node: nodes.getNodes()) {
            if (node.isNearby()) {
              nearbyNodes.add(node);
            }
```

[17] Remote procedure calls

```
            }
          }
        });
}
```

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:

```
private void startRequestingSensorData() {
  // send a request message to all nodes
  sendMessageToNearbyNodes("/start_requesting_sensor_data", null);
}

private void sendMessageToNearbyNodes(String path, byte[] payload) {
  for (Node node: nearbyNodes) {
    sendMessageToNode(node.getId(), path, payload);
  }
}

private void sendMessageToNode(String nodeId, String path, byte[] payload) {
  Wearable.MessageApi.sendMessage(googleApiClient, nodeId, path, payload)
      .setResultCallback(new ResultCallback() {
        @Override
        public void onResult(SendMessageResult sendMessageResult) {
          if (!sendMessageResult.getStatus().isSuccess()) {
            // perform exception handling
          }
        }
      });
}
```

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.

5.4.5. Receiving Messages

Apps running on the wearables need to implement the `MessageListener`[21] interface in order to get notified about incoming messages. These listeners have to be registered using the `MessageApi.addListener()` function.

```
@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`.

6. Evaluation

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 consecutive measurements and has been validated multiple times.

6.1. Setup

For the benchmarks, we created a `TimeTracker`[22] 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.

6.2. 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`[11] with the latest sensor data (wearable)
- Serializing a `DataRequestResponse`[12] containing that data (wearable)
- Transferring the data using the `MessageApi`[17] (wearable & mobile)
- Deserializing the `DataRequestResponse` (mobile)

In order to reduce the serialization overhead, we collect sensor data in `DataBatches` [11] 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 being less efficient or being 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` [12]. 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.

6.3. 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[18] because of its performance advantage compared to other well known libraries like gson[19].

There are benchmarks available that compare 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[20].

[18] https://github.com/FasterXML/jackson
[19] https://github.com/google/gson
[20] http://blog.takipi.com/the-ultimate-json-library-json-simple-vs-gson-vs-jackson-vs-json/

6.4. Battery Impact

Of course the heavy usage of the Bluetooth sensor and processing power brings along the cost of reduced battery life. Because battery power is very limited on mobile devices, we also benchmarked the power consumption of our app and the related system processes. Each measurement contains the consumed milliamp hours (mAh) during a one hour period.

We observed the processes with the largest battery impact, the following tables contain the consumption of:

- **Total**: The overall device
- **System**: Android System related packages
- **OS**: Android OS related packages
- **BT**: Bluetooth sensor
- **Screen**: Device display
- **App**: Sensor Data Logger package

To get some reference values, we observed the idle state first. Mobile and wearable device are connected and only casually used to reply to messages:

Total	System	OS	BT	Screen	App
163	27	20	13	32	0
173	30	22	15	52	0
173	34	19	13	39	0
160	27	25	13	40	0

Table 4: Battery impact, idle (in mAh)

We measured an average total battery drain of 167 mAh per hour. Assuming that the average battery capacity of currently available smartphones is ~2500 mAh, this is equivalent to 6.68% of battery life.

The Bluetooth consumption is noticeable because the wearable is connected to the phone and synchronizing notifications, although it is not streaming sensor data.

To benchmark how the app influences the power consumption, we requested 3-dimensional data from the accelerometer and gravity sensor with `SENSOR_DELAY_FASTEST` every 50 ms. The measured operations include:

- Transferring the data using the `MessageApi` (wearable & mobile)
- Deserializing the `DataRequestResponse` (mobile)
- Processing the `DataBatches` (mobile)
- Rendering the `Data`[10] (mobile)

Total	System	OS	BT	Screen	App
615	65	97	32	162	226
580	59	99	32	148	239
593	67	91	34	161	226
630	71	103	32	151	219

Table 5: Battery impact, with processing and rendering (in mAh)

The average total battery drain has increased by a factor of 3.6 to 605 mAh per hour. One can see that all observed processes consumed at least twice as much battery. As expected, the app itself is responsible for the largest battery impact with an average of 228 mAh.

However, a huge amount of power was required for the processing and rendering steps, which may be not required in every use case. Also, we forced the screen to be turned on at 20% brightness, which can also be avoided.

For the following measurements we excluded the processing and rendering steps and did not force the screen to be on:

Total	System	OS	BT	Screen	App
331	60	88	34	41	57
318	59	92	30	38	64
330	53	94	33	32	63
325	57	92	33	40	59

Table 6: Battery impact, without processing and rendering (in mAh)

Although all processes still require more power compared to the idle state, the average total battery drain has only increased by a factor of 1.9 to 326 mAh per hour. The `Data` processing and rendering steps were responsible for 73% of the apps battery drain.

Add visualization

7. Future Work

7.1. Open Research Questions

What could other researches work on?

7.2. Extensions

7.2.1. Channel API

In our implementation, we utilized the `MessageApi`[17] 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`[23] is also part of the `WearableApi`[13], accessible through the `GoogleApiClient`[14]. It should be used to transfer large files, just like the already mentioned `DataApi`[15]. In contrast to the `DataApi`, it doesn't synchronize `Assets`[16] across 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 yet.

7.2.2. Compression

In order to decrease the amount of data transferred with each `DataRequestResponse` [12], 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 of 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.

8. Conclusion

To achieve our project goal of a seamless authentication using nothing but the sensors from a user's devices, we need to transfer huge amounts of data every second. Our solution provides a performant jet battery friendly way of exchanging messages with data payloads between wearables and mobile devices. It allowed us to develop a product that fulfilled all our project requirements.

We also implemented our solution in an open source app[21] that is available for free through the Google Play Store[22]. It is capable of visualizing sensor data from the current mobile or a connected wearable device in real-time. This app can be used as a reference for related research and for demonstration purposes.

Fill a page, wrap up everything

[21] https://github.com/Steppschuh/Sensor-Data-Logger

[22] https://play.google.com/store/apps/details?id=net.steppschuh.sensordatalogger

References

[1] Romit Roy Choudhury He Wang, Ted Tsung-te Lai: *Mole: Motion leaks through smartwatch sensors*, 2015.

[2] Ethan Blanton Steven Y. Ko Lukasz Ziarek Yin Yan, Shaun Cosgrove: *Real-time sensing on android*.

[3] Google Developers: *Sensor manager*, July 2016. `https://developer.android.com/reference/android/hardware/SensorManager.html`.

[4] Rob Baldauf Waldhör Klemens: *Recognizing drinking adls in real time using smartwatches and data mining*, 2015.

[5] Google Developers: *Sensor*, July 2016. `https://developer.android.com/reference/android/hardware/Sensor.html`.

[6] Google Developers: *Sensor event listener*, July 2016. `https://developer.android.com/reference/android/hardware/SensorEventListener.html`.

[7] Google Developers: *Sensor event*, July 2016. `https://developer.android.com/reference/android/hardware/SensorEvent.html`.

[8] Google Developers: *Activity*, July 2016. `https://developer.android.com/reference/android/app/Activity.html`.

[9] Stephan Schultz: *Sensor data logger: Data package*, July 2016. `https://github.com/Steppschuh/Sensor-Data-Logger/tree/master/Android%20Source/datalogger/src/main/java/net/steppschuh/datalogger/data`.

[10] Stephan Schultz: *Sensor data logger: Data*, July 2016. `https://github.com/Steppschuh/Sensor-Data-Logger/blob/master/Android%20Source/datalogger/src/main/java/net/steppschuh/datalogger/data/Data.java`.

[11] Stephan Schultz: *Sensor data logger: Data batch*, July 2016. `https://github.com/Steppschuh/Sensor-Data-Logger/blob/master/Android%20Source/datalogger/src/main/java/`

`net/steppschuh/datalogger/data/DataBatch.java`.

[12] Stephan Schultz: *Sensor data logger: Data request response*, July 2016. `https://github.com/Steppschuh/Sensor-Data-Logger/blob/master/Android%20Source/datalogger/src/main/java/net/steppschuh/datalogger/data/request/DataRequestResponse.java`.

[13] Google Developers: *Wearable api*, July 2016. `https://developers.google.com/android/reference/com/google/android/gms/wearable/Wearable`.

[14] Google Developers: *Google api client*, July 2016. `https://developers.google.com/android/reference/com/google/android/gms/common/api/GoogleApiClient`.

[15] Google Developers: *Data api*, July 2016. `https://developers.google.com/android/reference/com/google/android/gms/wearable/DataApi.html`.

[16] Google Developers: *Asset*, July 2016. `https://developers.google.com/android/reference/com/google/android/gms/wearable/Asset`.

[17] Google Developers: *Message api*, July 2016. `https://developers.google.com/android/reference/com/google/android/gms/wearable/MessageApi`.

[18] Stephan Schultz: *Sensor data logger: Data request*, July 2016. `https://github.com/Steppschuh/Sensor-Data-Logger/blob/master/Android%20Source/datalogger/src/main/java/net/steppschuh/datalogger/data/request/DataRequest.java`.

[19] Google Developers: *Node*, July 2016. `https://developers.google.com/android/reference/com/google/android/gms/wearable/Node`.

[20] Google Developers: *Node api*, July 2016. `https://developers.google.com/android/reference/com/google/android/gms/wearable/NodeApi`.

[21] Google Developers: *Message listener*, July 2016. `https://developers.google.com/android/reference/`

`com/google/android/gms/wearable/MessageApi.MessageListener`.

[22] Stephan Schultz: *Sensor data logger: Time tracker*, July 2016. `https://github.com/Steppschuh/Sensor-Data-Logger/blob/master/Android%20Source/datalogger/src/main/java/net/steppschuh/datalogger/logging/TimeTracker.java`.

[23] Google Developers: *Channel api*, July 2016. `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