MOBILE AND SOCIAL SENSING SYSTEMS A.Y. 2021/22

These are my notes/transcriptions of the whole course, taken day by day-ish. Some things might be missing - since I was a student-worker during that period of time i followed most of the lectures online, most of them with the recordings - and i am missing the recording on the 5 may lecture :c
I integrated some of the missing things from the Avvenuti lectures with the huge help of the AMAZING notes provided by Tommaso Amarante. I also tried to follow and transcribe Vecchio, but transcribing lines of code is tricky, so I advise you to study that part by looking at the slides.
Have fun studying this subject!!

- Anna F

LECTURE 1: 01/03/2022 (Avvenuti)

From the register: Course syllabus, organization, assessment and introduction.

learning objectives of the course: have the ability to understand key principles and techniques related to the collection, filtering and analysis of information from mobile devices and social networks with a specific focus on data from physical and human sensors, develop the skills required to design and implement mobile and wireless sensing applications.
Prerequisites: operating systems, networking, programming skills, network based programming.
Lectures will address:

- mobile platforms: android
- non gps based localization techniques
- publish/subscribe, distributed hash tables
- human activity monitoring/recognition
- social media sources and networks, social bot detection
- humans as sensors paradigm

In laboratory sessions students will exercise with:

- programming smartphone based mobile applications
- social bot detection techniques

“When did the ideas behind the mobile pervasive system come along?”
1991, Mark Weiser, during one of their presentations, put on the table some very visionary ideas about the evolution of computer systems.
At that time we had personal computers, and most of them were not connected to the internet, which was mainly accessed by organizations.
They said that specialized elements of hardware and software, connected by wires, radio waves and infrared, will be so ubiquitous that no one will notice their presence.

“The most profound technologies are the ones that disappear. They weave themselves into the fabric of everyday life until they are indistinguishable from it.”
Technology should create calm, and help you do something else beyond computing.

Distributed systems are comprised of

- remote communications (protocol layering, RPC and so on)
- fault tolerance, which means that if you have redundancies and can connect those redundancies you are able to always provide service
- high availability through replication, rollback recovery etc, it is enabled by replication
- remote information access like file systems, databases, caching
- distributed security like encryption, mutual authentication etc

All these elements can be combined with:

- mobile networking
- mobile information access - i want to be able to have the same access to the network even when moving
- adaptive applications - the behavior may change automatically based on variations in the context
- energy aware systems
- location sensitivity

All these elements combined together will comprise mobile computing.

If these elements are added to mobile computing:

- Smart spaces
- invisibility
- localized scalability
- uneven conditioning

There is PERVASIVE COMPUTING.

MOBILITY:
Physical mobility is related to hosts and users, which can change their geographical location whilst retaining the ability to communicate and perform computation. It is a requirement, an input of our problem.
Logical mobility is related to the possibility to, in order to have a better performance or make the application reach data that is required, and means that application components or data can relocate at runtime. It is a design choice. There is no killer application for logical mobility, but web applications do use this. For example, when we run a client side script, there is logical mobility - js code embedded in html for example, first is on the server and in order to run it we need to download it on our machine.
The most extensive form of logical mobility is an application which has the ability to move from one node to another.

Mobile devices are usually battery powered, connected wirelessly to and through the interned or a private network through wireless LAN or WAN technology, and wireless connection ties the mobile device to centrally located information infrastructure and application software (aka The Cloud)

Pervasive means “starting from a mobile”. The goal of pervasive computing is to create an environment where the connectivity of devices should be unobtrusive, always available. Pervasive computing combines wireless computing, voice recognition internet capability,

artificial intelligence, in order to create an environment saturated with computing and communication capability, yet gracefully integrated with human users.

Pervasive computing: computing devices become progressively smaller and more powerful, we have a local body area network, main computer devices that act as servers for the local area, to both communicate with local devices and also with wider networks.

Context awareness: a pervasive computer system that strives to be minimally intrusive has to be context aware - cognizant of their user's state and surroundings, and must modify its behavior based on this information.
Of course, a user's context might be very complex, consisting of attributes such as

- physical location
- physiological state
- emotional state
- personal history
- daily behavioral patterns

Wearable sensors are located on smart devices which are potentially worn continuously throughout the day. This massive amount of information has led to an increasing interest in the health field.
A key challenge is represented by the reliable extraction of relevant patterns from collected signals. For instance, gait analysis applications should devise appropriate signal processing techniques to detect walking activity as well as to analyze and classify gait patterns.

LECTURE 2: 02/03/2022 (Vecchio)

From the register: Introduction to mobile and pervasive computing. Location-aware computing. Wearable devices and smartphone sensing. Introduction to Android. History and present. Android software stack. App development. Dalvik and ART. Development environment. Isolation between apps.

Programming with smartphones and smartwatches is quite different - memory and battery are two variables that have to be taken into account.
Nowadays, computing power is not a problem anymore.

Smartphones are a mix of communication devices (feasible through a variety of technologies such as wi-fi, cellular, bluetooth, NFC), computing devices (powerful processors) and a variety of sensors (position, temperature, ambient light, pressure etc).

Sensors usually available are GPS, microphone, ambient light, proximity, compass, camera.
Other sensors may be present, such as fingerprint, heart rate, pressure, temperature, motion sensing, UWB.

All kinds of information about the physical world can be used to make intelligent sensing apps, like apps counting burned calories by measuring physical activity.

For example, devices with tiny radars can implement a service that maneuvers the phone without touch involved. A gesture provides a raw signal that gets transformed into a solid signal. from those signals, there can be a gesture classification that detects a type of gesture. You can detect many signals to calculate distances and velocities, but accuracy depends on the time that passes between different detections.

Some more details on communication technologies that we can use on a smartphone: WiFi (which uses less energy than a cellular network), Cellular networks, Bluetooth (also low energy), NFC. These communication techniques also have different ranges.

Mobile computing: we mean the use of a smartphone while moving. We generally have a continuous connectivity through jumping from cell to cell in an invisible way for the end user.
At an application level, there are not so many differences. It is the user who decides to interact with the application and initiates activities, and the infrastructure only provides connectivity.

Location aware computing is when the location of the user becomes one of the inputs of applications. Different position means different output.
Not all mobile applications are location aware, though. It depends on design and implementation.

One of the main differences between mobile computing and traditional applications is energy. During the last decades, all resources increased significantly with the exception of energy.
Energy saving techniques:

- CPUs able to adapt to the computational needs (most of the time smartphone is idle)
- Turn off screen aggressively (screen requires a lot of energy)
- Application specific methods: in video streaming reduce resolution
- Make the user aware of remaining energy: with this battery level you can browse the web for 1h, or play music for 2h.

Pervasive computing: a synonym is ubiquitous computing, it is the expression used to describe what computing would become in the 21st century and expresses how computers become “invisible”.
The final goal is to assist humans in tasks, understand the user's intention and stay invisible.
It is the system that, usually while using sensors to collect information about the physical space, initiates the activity. User’s explicit input may not be required at all.

In pervasive computing, wearable devices can provide patients with personalized health data, which could assist with self-diagnosis and behavior change interventions.
Wearable devices can collect relevant information for health purposes. The quality of the information is lower than what you can get from medical equipment, but wearables can collect data for a longer period.

Examples of applications:

- heart rate can be measured with an oximeter built into a ring
- muscle activity with an electromyographic sensor embedded into clothing
- stress with an electrodermal sensor incorporated into a wristband
- physical activity or sleep patterns via an accelerometer in a watch

- a female's most fertile period can be identified with detailed body temperature tracking
- levels of mental attention can be monitored with a small number of non-gelled electroencephalogram (EEG) electrodes
- levels of social interaction (also known to affect general well-being) can be monitored using proximity detections to others with Bluetooth - or Wi-Fi - enabled devices

As an example, we will see a pedometer that shows steps just by looking at acceleration.

Smartphone sensing: many applications can be implemented using a smartphone equipped with sensors as a platform.
From the sensors we get the raw data. From the raw data we extract the features we need and we can operate on the data we have by using machine learning.
Action to be inferred is hand-labeled to generate training data, then sensor data is mined for combinations of sensor readings corresponding to action.

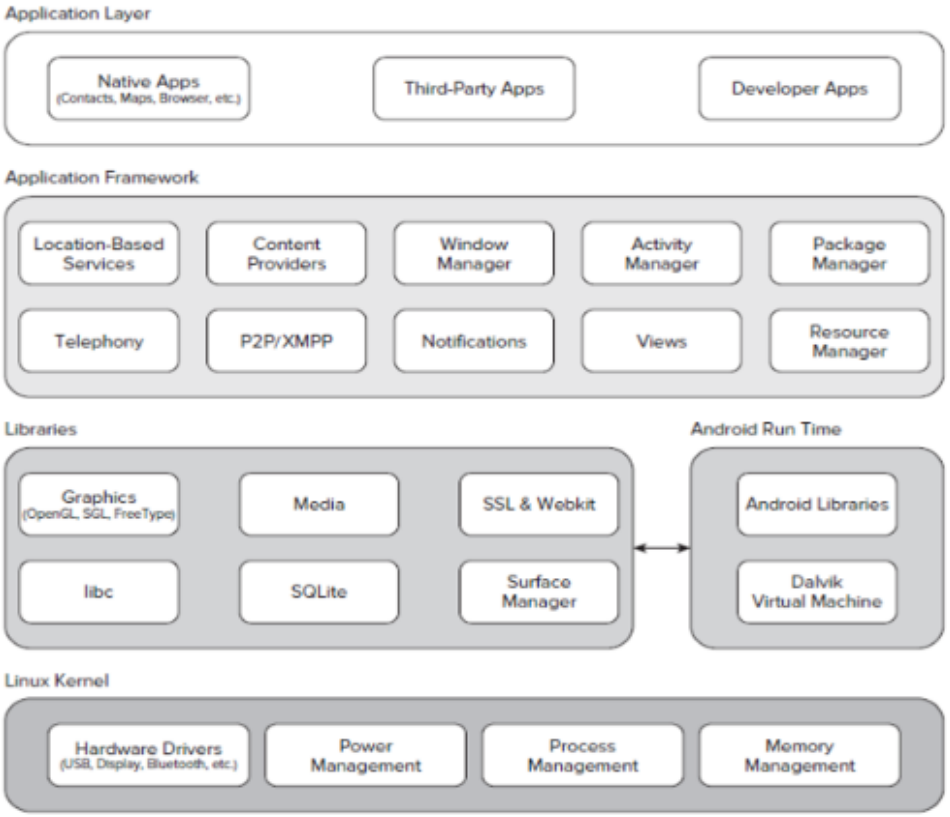
Other pervasive systems are Smart homes, Smart Buildings, Smart cities.

Now we talk about ANDROID.
Android is the world's most popular mobile operating system. It is owned, extended and maintained by Google, and there are many other companies backing the project: it is called Open Handset Alliance and it involves more than 80 companies.

History:
- 2005 Google buys Android Inc.
- 2007 OHA is announced and Android is presented
- 2008 Android SDK 1.0 is released
- 2010 Nexus One is commercialized
- 2012 800K+ Android devices activated every day

Android is also available for multiple devices and purposes:
- Smartphone (from the beginning)
- Tablet (support introduced in Android 3.0, now considered same as smartphones)
- TV (launch June 2014, Android 5.0)
- Auto (launch June 2014, Android 5.0)
- Wear (support introduced in Android 4.4)
- Glass (experimental project, now only limited distribution with selected companies)

Supported HW platforms:
- Supports ARM, x86, MIPS architectures (ARM from the beginning)
- Both 32bit and 64bit (the latter since Android 5.0)
- ARM/x86/MIPS or 32/64, transparent at the application level (unless you need native (C) code)



The one that we see in the picture is the software stack of the Android computing system. It starts with a linux kernel with some changes that have been included by Google, mostly related to the scheduler, which has been made to be more energy efficient.
Android Run Time is the managed runtime used by applications and some system services on Android. uses a set of libraries, in particular a standard C library (not the same that we find in Linux).
There are also other libraries. In general we use the application framework, where there are many libraries to let the developer interact with the phone. Applications are developed relying on this set of components.

The development model; there is a current release, and it gets inputs from upstream projects, and there is a parallel experimental branch which is then merged into the current release. There are also new branches created to create new versions.

Android app development: the tools are free, no specific license needed. The distribution of applications is not restricted, apps can be distributed as apk files containing programs and resources.
You can distribute via the app store, the web, or even via email.

Basically, you must write code in java - java and kotlin are basically the same and these two things are different because of commercial reasons. So, we must write code in Java and give it to the Java compiler. This gives us Java byte code, which must be processed by the Dx compiler. This gives us a Dalvik exe file which can be executed in the Dalvik VM. For google, you are writing Dalvik code.
The java we use for this is Java 8, with less Java libraries but a lot of android ones, like for graphical interfaces.

The Dalvik virtual machine is one where execution is interpreted. This approach though was not very efficient, so they switched to just-in-time compilation. So, the Dalvik VM sometimes translates into native code so that that is executed and there is no overhead due to interpretation, only some one time overhead for compilation. This was still not as efficient in terms of speed, so after Android 5.0, they changed it to Ahead of time compilation. So when you download the application, you download a Daltik executable file, but at installation time everything is translated into, basically, native code.
Installation now requires some more time, and you need some more space to store the executable files, but execution speed is better.

So, source is compiled into a dalvik executable, everything is put together with resources and we have the apk file. When we download it, everything is translated in a linux binary file and executed at full speed.

Some other advantages in this scenario is less energy spent.

We will focus on Android for many reasons. Mostly because it is available on 8 out of 10 devices, and development tools are free of charge and available for multiple OSs.
Having open source code is nice because you can check out how everything is done. Also, there is a larger user base and we already know the programming language (Java).

There are also some problems though: there are many different versions of Android, different devices, different APIs, different UI models, appearance, customizations, we cannot use the same functions everywhere. Some vendors introduce customizations to the UI and may provide additional libraries.
If we use native code, we must provide compiled binaries for the processor architectures we want to support.
Everything will be different depending on the device.

About Android Versions: there are ranges of compatibility with different versions, codename and API level.

Apps can run on the device, or on an emulator provided by Android Studio. This is not always better, but it depends on the application. The emulator more or less works like a real device.
Pros of the emulator: convenient execution of apps within the development environment, easy to test apps on various emulated devices and screen sizes without having to buy new devices.
Cons of the emulator: it is slower than a real phone, and there might be limited support for GPS, camera, video recording, phone calls, bluetooth devices, usb devices, battery level, sensors, etc.

Mechanism that android uses to separate applications. So, Android is based on the Linux kernel and is a multi user system. Each application is a different user of the system. This gives each application a separate environment. Each application is executed by a different VM, and every application has its own linux process.

Developing apps in Android separates the UI design code (XML) from the program (Java). We can thus modify the UI without changing the Java program.
For example, shapes and colors can be changed in the XML file without changing the Java program.
The UI can be designed using either XML or a drag and drop graphical tool (WYSIWYG).

Instead of a constructor, in Java, we have a method called onCreate(), called when an activity is created. An Activity is a screen of an application, and our activities are extensions of the AppCompatActivity class, which is also used on older versions of Android and can give us access to older functions.

For each activity, we can create the XML or use the drag and drop tool to create the graphical interface.

Let's look at the HelloWorld example: there are three most important files.
MainActivity.java contains the java code that defines the behavior that we want to be executed during the interaction with the screen (in this example we have an activity with a button and we want to click it)
activity_main.xml is the file that describes the UI in XML
AndroidManifest.xml is the file that includes a list of all the application components. In this case in particular there is only one. Every app has one.

These files are created automatically.

About the onCreate(): an app may have several activities, but the user interacts with one at a time. This method will be called by the system when the activity is started, and that is where you should perform all the initializations and UI setup.

In the activity_main.xml we can find different kinds of elements:
A layout is a container of UI elements. A TextView is a non-editable text box. We can have multiple containers, and the behavior of layouts and appearance of widgets can be edited through the use of properties.

In the Android Manifest we can see several nested elements. There is a package that shows us the Java package. Then our application is set as a component where all the other components are contained.
There is the tag for the activity as well, and also other properties regarding the application.

Project structure: there always is a java folder and a res folder with all resources, like XML and other kinds of elements, even several subfolders for images, general purpose files and other static resources we can embed in an Android screen.

AVDs are android virtual devices. We can use the AVD manager to create as many as we want to, and set up many different properties about them.

SDK manager allows us to install build-tools, system images, Android APIs, Google proprietary APIs, extras like Google USB driver, Google play services etc.

LECTURE 3: 08/03/2022 (Avvenuti)

From the register: Wireless (non-GPS) Localisation in Cyber-Physical-Systems: definition, taxonomy, approaches. Anchor-based localization: range-based, range-free approaches. Range-based localization techniques: range estimation: RSSI

Localization techniques, more precisely we will talk about wireless localization techniques.
Two types of distributed wireless systems:

1. Cyber - Physical systems: they are systems where in general you have some type of mechanism or environment which is controlled by a computer. It is a distributed, networked, embedded system. In such a system, the main characteristic is that physical and software components are deeply tied together. You can sample and sense data from the environment, for a long time. Another interesting feature is that they tend to "disappear", they are embedded.
2. Wireless Sensor Networks: they were the first implementation and deployment of cyber physical systems. It is built on any number of nodes, the characteristics of each node, besides being a computing system, it has many sensors in order to connect it to the environment and to get data from it. WSNs's nodes are all wirelessly connected between them, and most of the time each node has a battery. a WSN though is an instance of a CPS.

We want to be able to localize, in CPSs, not only the HW components, but also the events. If something happens, we want to be able to give these events some coordinates in order to understand and use the positions where the events occur.

Localize, in general, means to be able to produce coordinates that refer to a positioning system, and those coordinates can be used by the application for location aware services, or to support network services (report region of an event, track objects, evaluate the coverage of a network or assist with routing).

When we talk about localization systems, we need to do a taxonomy, because during the last years we saw a proliferation of approaches to these algorithms.

Localization systems consist of two main blocks:

- the set of deployed nodes
- the localization algorithm

Deployed nodes may have different states:

- Unknown: or target, they have no information over their position
- Beacon: beacon nodes are also called landmarks or **anchors**, and they are those that already know where they are, their position. This can be done by manual placement of the nodes or through GPS reading
- Settled: when a node succeeds in determining its estimated location

At the startup of the localization algorithm, nodes can either be in state beacon or unknown. We might have systems in which settled nodes start behaving like anchors.

Anchor based methods: some of the nodes deployed into the environment we know the position of, others we don't. The higher the density, the more accurate the localization algorithms. Unfortunately, it is not always possible to have an anchor in a system.
We usually evaluate the distance between targets and anchors by using simple geometric relationships, like trilateration (range based) and centroid (range free, no need to estimate distance from anchor to target)

Localization algorithms aim at finding the location or position of unknown nodes. Among the algorithms that can be used, there is of course the Global Positioning System which is very intricate and, from experience, is able to determine a position accurately. Unfortunately, it is not always possible to use it, for example because it does not work efficiently indoors, and for energy consumption reasons. GPS is going to consume a lot of energy to compute positions. So, while it is the best technique to evaluate and estimate the location, we cannot always use it.
In general, ways to determine the location depend on the objective of the cyber physical system.

Alternatives to GPS exist, and those techniques in general will exploit first the sensing capacity of each component, or/and wireless communication capabilities of CPS components. We will see that wireless communication can be used to estimate the position of something while avoiding GPS.
For example:

- use the Received Signal Strength (RSS) or propagation delay to infer the distances/angles to some reference nodes and then estimate the location through simple geometric computations - when a wave goes through space it loses energy and we can exploit this property to calculate distance
- estimate distances by exploiting the difference between heterogeneous waves propagation properties (like the time that takes to receive a signal from source to destination, propagation speed, or reception time difference between acoustic and radio waves)
- range free localization determines unknown node position based on proximity/connectivity information, less accurate but it works

Topology of a localization system: where and how is the location of a given node calculated?
Four different system topologies:

1. Remote positioning: the location is computed at a central base station, where anchor nodes collect transmitted radio signals from the mobile node and forward them to the base station. The base station itself computes the estimated position of the mobile node based on the measured signals forwarded by the anchors;
2. Self Positioning, we have unknown nodes that after having received a number of messages from anchors can compute autonomously their position (GPS!!!);
3. Indirect Remote PositioninG - like self positioning but the position is first calculated by the mobile node and then sent to a base station through a wireless back channel;
4. Indirect Self Positioning: like in the remote positioning, the location of the mobile node is computed at the base station, then the base station forwards it to the mobile node through a wireless back channel

Coordinate system:

- Absolute - in relation to anchors
- Relative - in relation to other nodes
- Physical - point in a system
- Symbolic - conceptual coordinate

Centralized versus localized:
Centralized means that we have a central device that estimates the location of unknown nodes based on the signal measurements. Better accuracy but not easy to scale, and when we do not have enough anchor nodes we need wireless communication systems that are multi hop, and when we use that we will consume energy which is not the best.

Localized: used by self positioning systems, each object estimates its location by collecting from the anchor nodes in the neighborhood. Sometimes they require iteration between nodes, much simpler though.

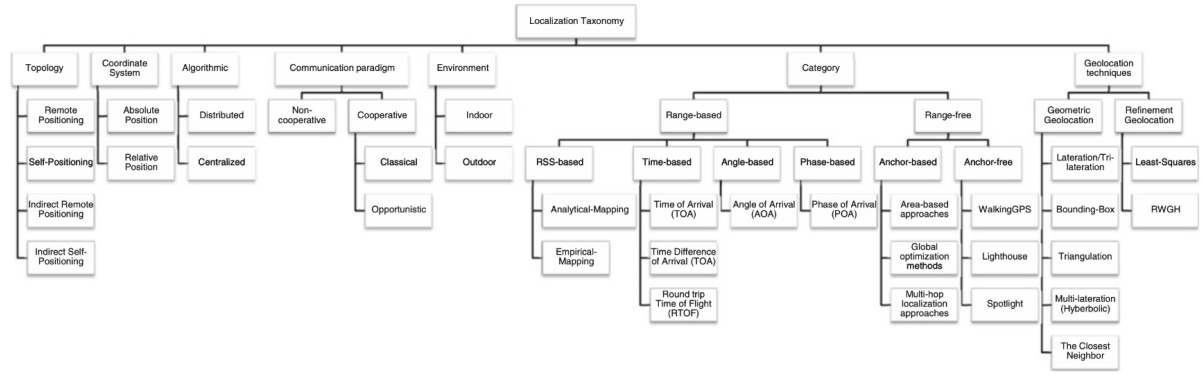
Finally, if we want to measure the performance, we need different metrics such as:

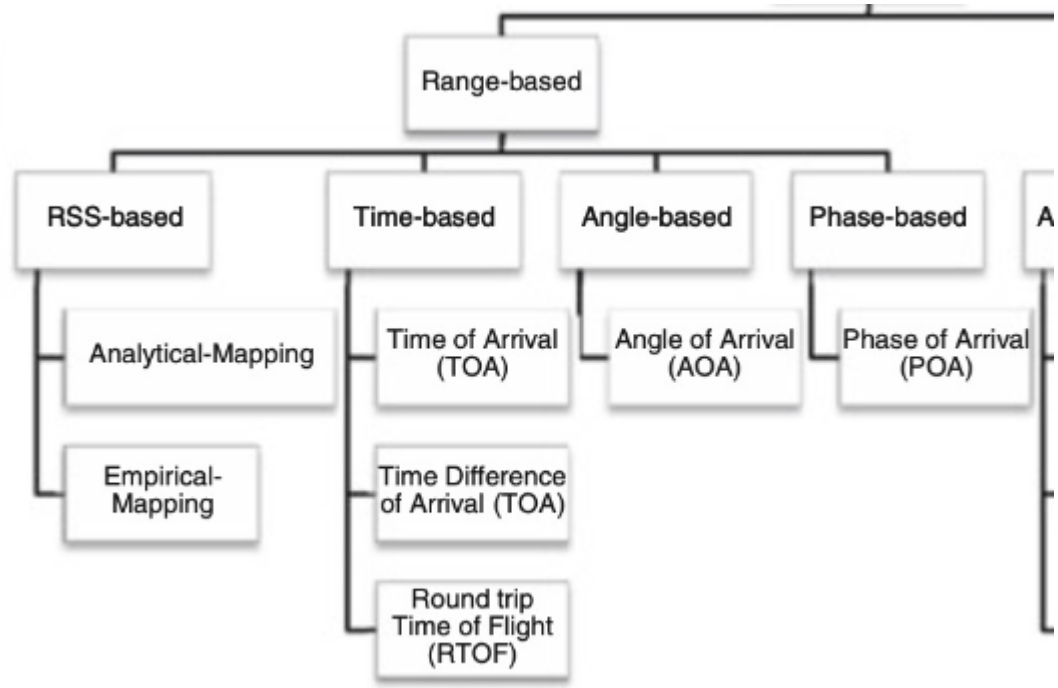
- Accuracy - mean distance error
- precision - variation in algorithm's performance over many trials
- complexity - can be attributed to hardware, software and operation factors, and usually location rate is an indirect indicator for complexity
- robustness - the system's capability of working when some signals are not available
- scalability - the positioning performance degrades when the distance between the transmitter and receiver increases. a location system may need to scale on two axes: geography and density
- cost - infrastructure, maintenance, additional nodes…

Communication paradigm:

- Non cooperative - the communication is restricted between unknown nodes and anchors (and there is no communication between nodes with unknown locations) so high density of anchors or long-range anchor transmissions are needed to ensure that each unknown node is within the communication range of at least three anchors
- Cooperative - allows communication between unknown nodes. anchor density is alleviated but more processing is required.
- Opportunistic - we observe the node while it is doing something other, or we piggyback a node.

Now, we are going to discuss some examples of range-based taxonomies.





So, in range based systems we need to calculate a distance from the device to a sort of an anchor. We usually exploit information that comes from the wireless communication subsystems.

The characteristics are that location discovery mainly consists of two phases. The first is called ranging phase, in which a node must either determine or estimate the distance between some anchors, and the estimation phase where, using geometric relationships and information coming from the first phase, the position can be estimated.

There are several distance measuring methods, like strength of wireless signal (which uses RSSI readings, that is to say Received Signal Strength Index), or Time of Arrival (which is used with radio, acoustic, ultrasound, and we are required to get the timestamp of departure and arrival and the type of signal's speed of propagation) or even the Angle of arrival, which is very useful with directive antennae or arrays of antennae.

The estimation phase can involve different techniques. One of them is trilateration which, by using noisy measurements, can let us determine the location of a point by measuring the

distance from reference points - we can seek for the intersection of the circles starting from the three points where the anchors are positioned, and with the distance as radium. We could use self positioning schemes with non cooperative communication (self positioning because target nodes wait from messages coming from anchors, so anchors continuously send beacon messages where the payload is the coordinates of the beacon itself, and non cooperative communication because the three beacons do not need to know about each other, and the position is locally calculated by the point).
Triangulation lets us determine the location by measuring angles between three reference points. If we know the angle at which we receive the message, we can place our position. The smaller the triangle is, the more accurate the estimation will be. That's why the higher the density of the beacons is, the more accurate the positioning will be.
If the density is high we can have multilateration, which considers all available beacons. We can build polygons and seek intersections between them.

Received Signal Strength is a relationship between the power as a function of the euclidean

$$P_r(d) = \frac{P_t G_t G_r \lambda^2}{(4\pi)^2 d^2}$$

distance between receiver and transmitter. d is the distance, whilst Pt is the power of the signal at the transmitting node, Gt and Gr are constants called "gain", numbers that represent properties of both systems' antennae, and lambda which is another constant.
If we know Pt and Pr we can compute d. Pr is a number that we can easily get from our communication interface API. The parameters employed are site specific, the signal is weaker and more exposed to errors and path-loss models do not always hold in a real environment due to multipath fading and shadowing. This formula holds better in free space and in an ideal environment.

Site specific means that before deploying our system, we must go on field and take some measurements in order to understand how radio signals propagate in the specific environment in which we are in.
The power can be generically related to a constant X divided by the distance r to the power of a value n. This is an empirical model that we can make knowing that radio frequency signal attenuation is a reverse proportional function of distance.

Time of Arrival is based on the time relation and it is an ideally very straightforward method. We only need the propagation speed and the time of departure of the signal. The problem is that if we use a radio signal the propagation speed is very close to the speed of light, and if we do not have extremely precisely synchronized nodes even a tiny error in measuring time is scaled in a dramatic way and makes up for a huge error in distance.
If we do not have good synchronization, we can use the time difference of arrival.
For ToA, GPS can be a good synchronization. We cannot always use GPS though. Time synchronization is not a problem for the satellite constellation that rules GPS, because they have atomic clocks. They also use trilateration.

LECTURE 4: 09/03/2022 (Vecchio)

From the register: GUIs in Android. Widgets. XML- and Java-based development of GUIs. Texte fields, buttons, and events. Other widgets. Layouts. Common widget and layout attributes. Resources: strings, colors, dimensions, styles, themes. Animations, menus. Alternative resources. Multi-language applications. Using resources in Java and XML. Data-driven layouts: ListView. Components of Android apps. Activities. Fragments. Services. Content providers and broadcast receivers.

Android GUIs and the main ideas behind its development.
Let's start with some GUI elements, which are the typical elements of an user interface. All widgets are subclasses of the View class.
Examples are:

- Buttons
- EditText (editable texts)
- TextView (read only text labels)
- CheckBox
- RadioButton
- ToggleButton (for choosing between two states)
- RatingBar
- Spinner (lets you select an item from a list to display in the textbox)

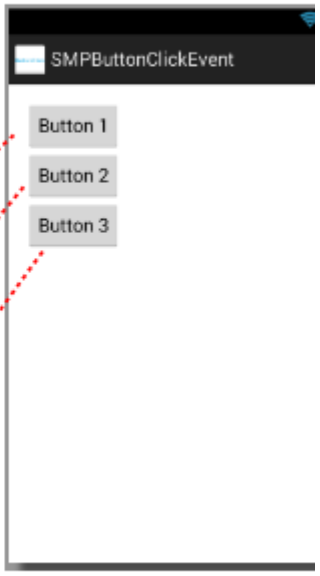
in Android, the elements that appear on the screen are defined by XML files that describe the user interface. To declare a widget type you just put one in a tag with the type and a list of attributes.
Example of a TextView element:

The actual user interface is described by an XML file but the best choice is to use the drag and drop tool, and then set properties.
Something we have to do with all elements is to define an ID to use in our code in order to modify the element, and interact on the code side with the element.

Just as examples, different widgets.

Buttons can be customized in different ways. Button is a subclass of TextView. We can receive the events generated by the buttons in an interesting way. We can use different mechanisms.
The first is used for button 1 and 2 in the picture, the second in button 3.

```
public class MainActivity extends Activity implements OnClickListener {

        final static String TAG = "MainActivity";

        @Override
        protected void onCreate(Bundle savedInstanceState) {
                super.onCreate(savedInstanceState);
                setContentView(R.layout.activity_main);
                Button b1 = (Button) findViewById(R.id.button1);
                Button b2 = (Button) findViewById(R.id.button2);
                b1.setOnClickListener(this);
                b2.setOnClickListener(this);
        }

        @Override
        public void onClick(View v) {
                if(v.getId() == R.id.button1)
                        Log.i(TAG, "You pressed button 1");
                else if(v.getId() == R.id.button2)
                        Log.i(TAG, "You pressed button 2");
        }

        public void myMethod(View v) {
                CharSequence l = ((Button) v).getText();
                Log.i(TAG, "You pressed button labeled " + l);
        }
}
```

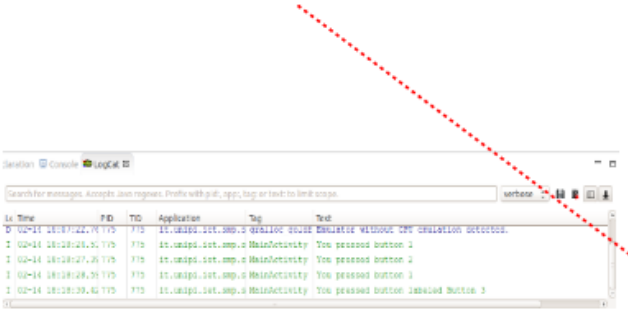


As we saw, MainActivity extends Activity (which is a screen in an application). onCreate() is executed by the operative system when the activity is started, actually when it is created. Within this method, the setContentView shows the graphical user interface on the screen defined by an XML layout file, the “activity_main”.
Then, we must connect the buttons to the methods - first we find a java reference to the button object (findViewById where we give the ids to the buttons, and we convert the result of this function to a Button Object from an instance of view) and finally we specify which is the listener for the onclick event. The activity specifies itself as the receiver object. For this reason, the activity implements the onclick listener interface, that includes a single method we are forced to provide an implementation for - that is, onClick().
When button 1 or 2 is pressed, onClick is executed.
Log is a class that provides some information and has some methods which correspond to different levels of verbosity. i is for info. all the messages that we log by the means of this mechanism are propagated from the real device to the log console that is available on android studio.
These methods let us provide a TAG, easy to sort all the messages received by the log.
There is also another method - which we see implemented in button 3.
The second mechanism is probably simpler. We only need to specify in the XML which method should the button do if it is clicked. We call this mechanism XML based. Here in the picture there are also some constraint on the method.

Button

The method must

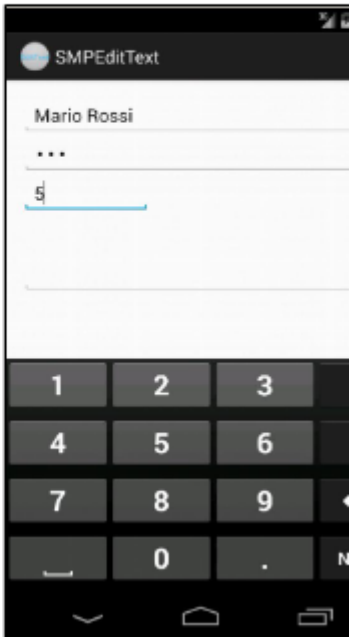
- be public
- return void
- accept a single *View* parameter



```
...
<Button
    android:id="@+id/button1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentTop="true"
    android:text="Button 1" />
  <Button
    android:id="@+id/button2"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/button1"
    android:layout_below="@+id/button1"
    android:text="Button 2" />
  <Button
    android:id="@+id/button3"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignLeft="@+id/button2"
    android:layout_below="@+id/button2"
    android:text="Button 3"
    android:onClick="myMethod"/>
```

Another widget is images. We can show an image via ImageView or put it on a button via ImageButton. Images are resources of our application. android:src attribute is used to select the path in the drawable folder. Attributes of the images define size etc.

```
...
        <EditText
    android:id="@+id/nameField"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="textCapWords" >
    <requestFocus />
</EditText>
<EditText
    android:id="@+id/passwordField"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:inputType="textPassword" />
<EditText
    android:id="@+id/numberField"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:ems="6"
    android:inputType="number" />
<EditText
    android:id="@+id/multiLineField"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:minLines="3"
    android:inputType="textMultiLine"
        />
```



EditText is a text box used for user input, and we can set the type of keyboard and input allowed with android:inputType (we can see the example in the picture next to this text)

Other widgets which we are not going to examine in detail but deserve to be quoted include CheckBox, RadioButton, ToggleButton, Switch, DateView, ProgressBar, RatingBar.

```
...
<Spinner
    android:id="@+id/spinner1"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_alignParentLeft="true"
    android:layout_alignParentTop="true" />
...
```

```
<?xml version="1.0" encoding="utf-8"?>
```

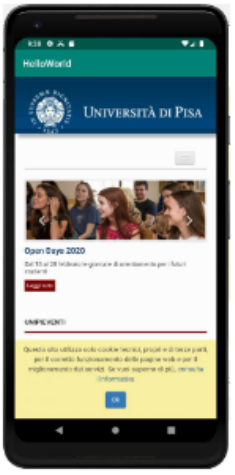
Spinners are something like a drop down menu. It lets us select one value from a list of values which are provided to it as an XML resource. We can fill this list of possible values by using another resource file, so the idea is that the list is in another file and we can create a spinner object and connect it with the list of values. There is an example of these two parts in the picture here on the side.

Webviews is another complex component. It is basically a rendering widget to visualize HTML pages, to show some documentation to the user or to show a website that is already built and there is no need to reimplement everything in the application. You just need to create the component, and with a single method you just have to specify the web page that you want to show to the user.

- Shows webpage

```
public class MainActivity extends AppCompatActivity {

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

        WebView myWebView = (WebView) findViewById(R.id.my_wv);
        myWebView.loadUrl("https://www.unipi.it");
    }
}
```

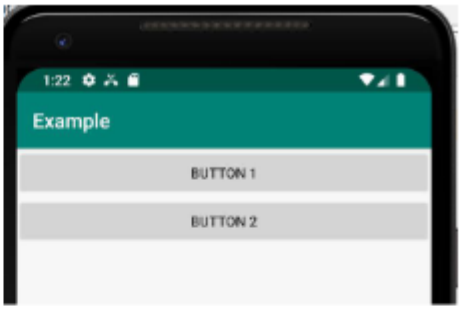
```
<WebView
    android:id="@+id/my_wv"
    android:layout_width="match_parent"
    android:layout_height="match_parent" />
```



In the onCreate a layout file with the webview is inflated, then a reference to the webview is obtained and then, by calling loadUrl method the page is loaded and shown to the user.

The elements that we just saw are basic elements, and we need something to unite them to create a complex interface. This is why we have layouts, which are containers of other widgets. They can be nested and can determine the position of a widget on the screen. They are XML files stored in res/layout, and Android provides many of them, already defined by the libraries.

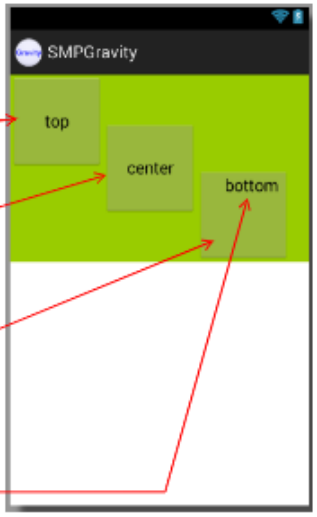
Linear Layout: all elements are placed linearly, either on the x or on the y axis. One of the attributes determines orientation, vertical or horizontal.
The example in the picture has 2 buttons.
There are 2 important attributes, almost always defined, called layout_width and layout_height.
We see two different values that they are set to and that have a particular meaning: “wrap_content” means that the widget/layout will be as wide/high as its content (e.g. text) whilst “match_parent” means that the widget/layout will be as wide/high as its parent layout box. These values can be applied to almost all widgets and layouts, either manually or by using AndroidStudio’s attribute editors, both for height and width.

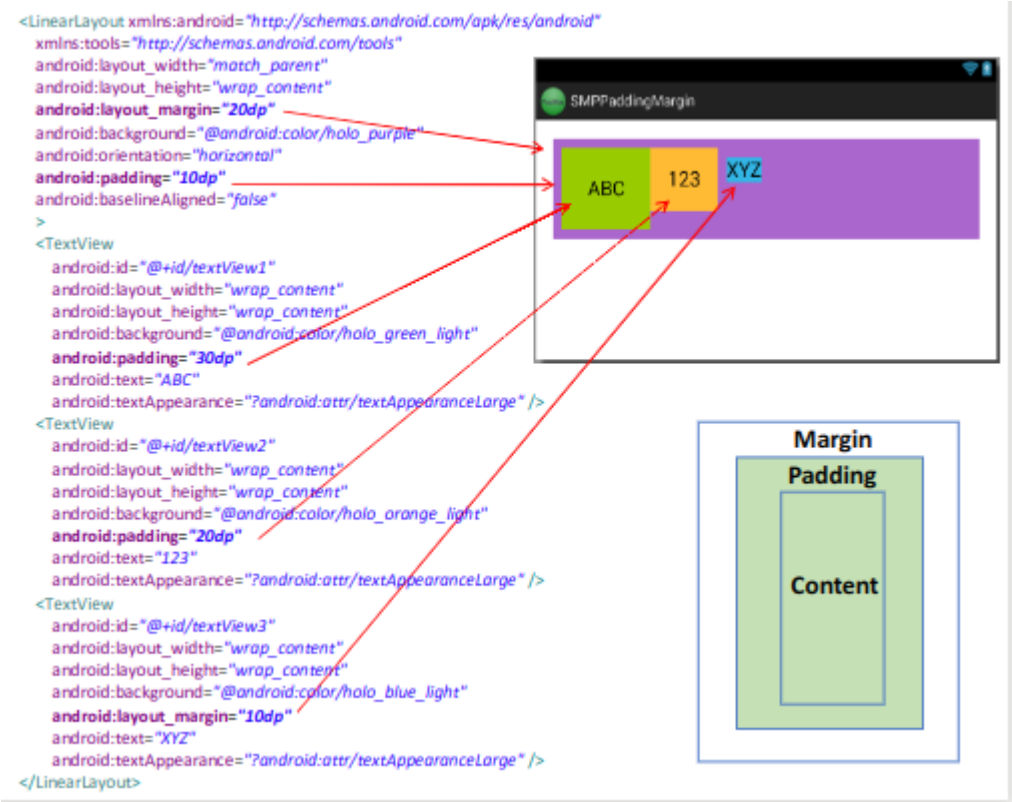


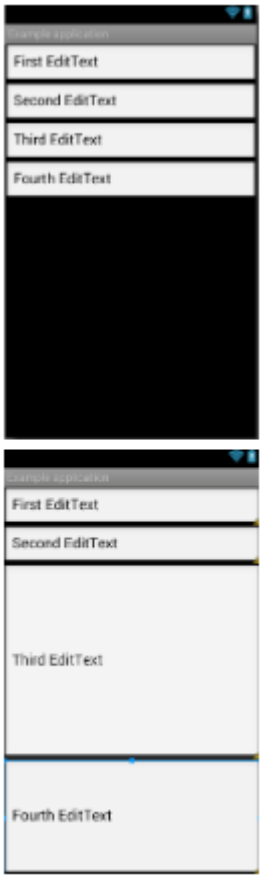

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:layout_width="match_parent"
    android:layout_height="match_parent"
    android:orientation="vertical">

    <Button
        android:id="@+id/button"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 1" />

    <Button
        android:id="@+id/button2"
        android:layout_width="match_parent"
        android:layout_height="wrap_content"
        android:text="Button 2" />
</LinearLayout>
```

All the elements are also characterized by a border, a padding and a margin (like HTML/CSS), as well as gravity (which determines how content is aligned within a view) and layout_gravity (how a view is aligned within a container. Here are the pictures with examples.

```
<LinearLayout xmlns:android="http://schemas.android.com/apk/res/android"
  xmlns:tools="http://schemas.android.com/tools"
  android:layout_width="match_parent"
  android:layout_height="200dp"
  android:background="@android:color/holo_green_light"
  android:baselineAligned="false"
  android:gravity="left"
  android:orientation="horizontal"
  >
  <Button
    android:id="@+id/Button01"
    android:layout_width="100dp"
    andr      oid:layout_height="100dp"
    android:layout_gravity="top"
    android:text="top" >
  </Button>
  <Button
    android:id="@+id/Button02"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:layout_gravity="center"
    android:text="center" >
  </Button>
  <Button
    android:id="@+id/Button03"
    android:layout_width="100dp"
    android:layout_height="100dp"
    android:layout_gravity="bottom"
    android:gravity="top|right"
    android:text="bottom" >
  </Button>
</LinearLayout>
```





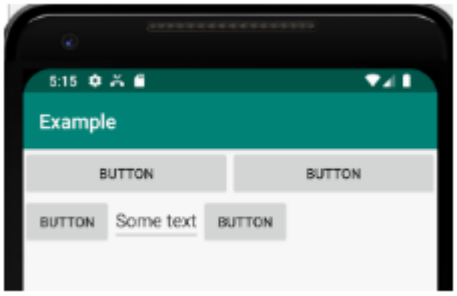


A weight can be assigned to contained elements as well. The default weight is 0 for each element and where there is an empty space, it is assigned to the children in the proportion of their weight.
For example, in the picture next to this text, if we do not give any weight the situation for a linear layout stays like in the upper picture, whereas if we give to the third edit text a weight of 2 and to the fourth a weight of 1, the situation looks like the downward picture.

Since in a smartphone the screen has limited space, scrolling is quite common. It can be done by using containers that allow scrolling functionality. We can scroll vertically or horizontally.
We just have to place a single element within the ScrollView (or the HorizontalScrollView).
Rules: only one direct child View, then the child can have many children.

ConstraintLayout: this is a very frequent layout, where we define the position of the elements as a set of constraints that are relative to the container's border and other contained elements.
The constraints are rather complicated to write, so we rely on the drag and drop tool to show us the constraints in a graphical way.
It is faster to draw such a system, it reduces the need for nested layouts and keeps the hierarchy flat - which is also more efficient. It is also flexible.
If you do not put enough constraints, Android Studio will give warnings.

The old version of this layout was called RelativeLayout. This layout is adaptable to different screen sizes and formats.



Nesting layouts is possible, if we want to define more complex UIs.
For example, in the picture next to this text, there is a vertical layout that contains two horizontal linear layouts. The first one has 2 buttons, the second one 2 buttons and a text.

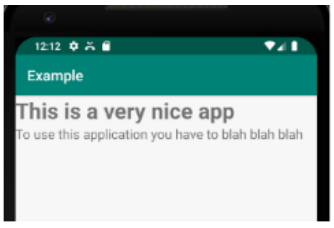
There are also sometimes where it can be necessary to create and organize the elements of the user interface using java code, in particular when the interface must be dynamic. There is a Java class for every element that we saw so far, and there are also Java classes corresponding to the constraints and sizes that we can give to every element, and what we can do is just create instances of these classes.
Using the official documentation is always useful.

Let's now look at the main ideas behind resources. We already saw possible resources, but the idea is to understand how we can use these resources in XML and Java.
Resources are what is needed for the execution of our application behind the code itself,
In particular, in android there are some predefined resources: strings, dimensions, layout files, menus, images, audio files, databases etc.

Strings are resources that are useful, because they allow us to decouple the content of the GUI from the structure of the GUI. The structure is defined by an XML layout file, the content can be defined externally - this way we can provide different versions of the applications with different strings, but the same structure (so, maybe, different translations).
All string resources are in a single file called strings.xml which can be found within the values folder within the res folder. Different strings defined in this xml file need to have a name, which will be the name referenced in the layout file.

Strings

- Then can be used in any other XML file



Layout file

```
. . .
<TextView
    android:id="@+id/textView1"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/my_screen_title"
    android:textSize="30sp"
    android:textStyle="bold" />

<TextView
    android:id="@+id/textView2"
    android:layout_width="match_parent"
    android:layout_height="wrap_content"
    android:text="@string/my_instructions"
    android:textSize="18sp" />
. . .
```

strings.xml

```
<resources>
    <string name="app_name">Example</string>
    <string name="my_screen_title">
       This is a very nice app</string>
    <string name="my_instructions">
       To use this application you have
       to blah blah blah</string>
</resources>
```

This can be done for the application name as well, in the AndroidManifest file.

Some basic HTML formatting can be included in string resources, and new strings can be added either by editing the strings.xml file or using the tools provided by AndroidStudio.
To translate an application, we just need to provide versions of the strings.xml file in different languages. It can be done in AndroidStudio with its interface.

Actually, languages are part of a larger mechanism which is the possibility to select alternative resources, to adapt for different languages, screen sizes, settings and day/night mode. This can be done with strings, images, sounds etc. We must, though, have default resources to use regardless of the device configuration, or when there are no alternative resources. This can be done by manually organizing the folders or by using AndroidStudio.

These are the main resource types:

- simple values (strings, colors, styles, dimensions)
- drawables
- mipmaps
- layouts
- animations
- menus
- raw resources

And they are all stored in subfolders of res/

Colors are expressed in RGB + transparency, whilst dimension can have different units (dp, mm, in, px, sp) it depends on the device and on the item.
We can use them in widget properties by selecting the resource in the XML (or in the AS Tool).

Styles are collections of properties that specify the look of widgets, we can define them in an XML resource. The same style can be applied to multiple elements. Styles need to have a name and can be organized in a hierarchical fashion. If a style inherits another style, it just needs to provide the difference with the otherwise inherited properties.

Themes are styles applied to a whole activity, or application. We can define new themes by using android:theme in the android manifest file. This allows us to change the appearance of the whole application. Some themes are already defined, in @android.style

Drawables are images. Several formats are supported and we can select alternative versions. For example, we can select different images for the different dpis that we want to support. Vector images can be scaled without problems so we should not have resolution problems. The resource identifier for a Drawable resource is the file name without the extension.

At runtime, Android chooses which resolution/directory depending on phone resolution, and there is a tool called Image Asset Studio which generates icons in various densities from original images.

There are also animations, frame by frame, defined by means of XML files. We must provide the list of images and then it cycles, or list the modifications we want to make to a View.

Menus are provided also as XML files that define their content and structure. We can create them in Android Studio, but we must write some code to “inflate” them in Activity - the method that inflates them is specified in onClick().

```
public class MainActivity extends AppCompatActivity {

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

    @Override
    public boolean onCreateOptionsMenu(Menu menu) {
        getMenuInflater().inflate(R.menu.my_menu, menu);
        return true;
    }

    public void mymethod1(MenuItem v) {
        Log.i("Example", "First option");
    }

    public void mymethod2(MenuItem v) {
        Log.i("Example", "Second option");
    }

}
```

```
<menu xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:android="http://schemas.android.com/apk/res/android">
    <item
        android:id="@+id/first_item"
        android:onClick="mymethod1"
        android:title="First option" />
    <item
        android:id="@+id/second_item"
        android:onClick="mymethod2"
        android:title="Second option" />
</menu>
```

It is an approach similar to the one for the buttons.

So, resources can be used from Java code or be referenced from within other XML resources.

There is a class, generated automatically by Android Studio, called the R class, which functions as a connection between Java code and resources (either in XML or provided as files). In particular, this class contains a number of internal static classes with names corresponding to the possible resource types. These classes contain constant values.

In java code we must obtain a resource by first actually finding it by using the findViewById() method and providing the resource identifier. We can extract information from the resource, after that.

Actually, to identify a resource we can either use the ID or the name of the resource. The name of the resource can be the name of the file (like for menus) or the name of the specific resource, for simple values like strings.

In XML things are more or less the same. We can use resources as attribute values in other XML resources by writing
attribute = "@resourcetype/resource_identifier"

Resources can be divided in project's and OS's. The operative system has many useful resources. To use Android resources we should use:

- android.R in java code
- @android:

rather than the application-specific R class.

Now, let's talk about data-driven layouts.
Let's talk about a simple example about a component that can be included in a user interface where the data is not statically defined in terms of XML files, but comes from somewhere else.
So, we saw LinearLayout, ConstraintLayout, RelativeLayout - every element is hard coded. What if we want to dynamically populate a user interface? The layout must be dynamically generated.

A ListView is a list of values with vertical scroll, A GridView is a grid with items arranged in rows and columns, ViewPager lets us swipe to show a new item and can be associated with a TabLayout.
We must define a component that retrieves the data from the sources and feeds it to the layout. This component is called an Adapter. There are several types.

Most common adapter types are CursorAdapter, which reads data from a database, and ArrayAdapter which reads from a Java array or a resource like a XML file.
Basically, an adapter reads the data from the data source and creates widgets to populate the list (in this case we are talking about a ListView with an ArrayAdapter but in other situations the thought process is similar).
More in detail, the adapter reads in data (list of items), creates Views (widgets) using layout for each element in data source, fills the containing layout (List, Grid, etc) with the created Views. Child Views can be as simple as a TextView or more complex layouts / controls; simple views can be declared in a layout XML file (e.g. android.R.layout).

For example, if we want to create a ListView with an ArrayAdapter, we define how the ListView is built and keep it empty at the beginning.
Then, in the MainActivity class (which extends AppCompatActivity and implements OnItemCLickListener) we put the array.
Then, in the onCreate, first the user interface is inflated. Then, we obtain a reference to the ListView by using the findViewById method. Then we set the adapter for the ListView. The adapter is an instance of an ArrayAdapter. We have to provide a number of values as input. One of them is an instance of context. There are many methods, in Android, that require an instance of context to operate. An activity is an instance of context and might provide itself as the context. The second argument of the setAdapter is a layout file that defines the

appearance of the single elements of the list. Some kinds of these layout files are provided by the system. Finally, we give the array.

Then, there is a listener attached to the listView. It is the activity itself so that when an item of the list is clicked the method runs. This method has the position value of the element in the array. This is used in a Toast (which is a small piece of text that appears onto the screen and automatically disappears after a few seconds). It is possible to specify how much time the Toast stays visible.

Now, we are going to see the types of components that are used to design an Android Application. In particular, we have to start by considering that an application on a smartphone operates in a different environment than applications in a pc. The user switches between apps quickly, and there are events coming from outside.
There is no main(). An app is a collection of components derived from OS classes.
There are four types of app components: Activities, Services, Broadcast receivers, Content providers.
We do not have to use them all necessarily, and we can have multiple of each component.
The basic idea behind this model is that an application can be organized as a set of loosely coupled components. That is to say, a component is able to trigger another component's execution even if they do not belong in the same application. This can be done also when the two components do not know each other: there are just messages sent at runtime.
When the system starts a component, it starts the process for that application (if it is not already running) and instantiates the classes needed for the component.
For example, if you want the user to take a picture, there is another app that does that and your app can use it, instead of developing an activity to capture a photo yourself.

Recap on activities: they are analogous to windows in desktop applications. They fill a screen, and are used for a task. Apps typically have at least one activity that deals with UI, and typically have more activities.

The problem is that there are many different screen sizes, and that makes us need to try different positionings for the components inside an activity.

Fragments are UI building blocks that are contained inside one Activity. They can be arranged inside activities in different ways depending on the devices, and enable apps to look different.

Activities are short-lived and can be shut down anytime. Services are not provided with a UI and keep running in the background. Typically an activity will control a service, starting, pausing or getting data from it. Services in an App are sub-classes of Android's Service class.

There are also managers that can be used to provide high-level functionalities or access to HW, like LocationManager, ClipboardManager, FragmentManager, AudioManager or DownloadManager.

There are also Google Services: these are not components but services in the classical way, not necessarily components of our application but could be external entities providing

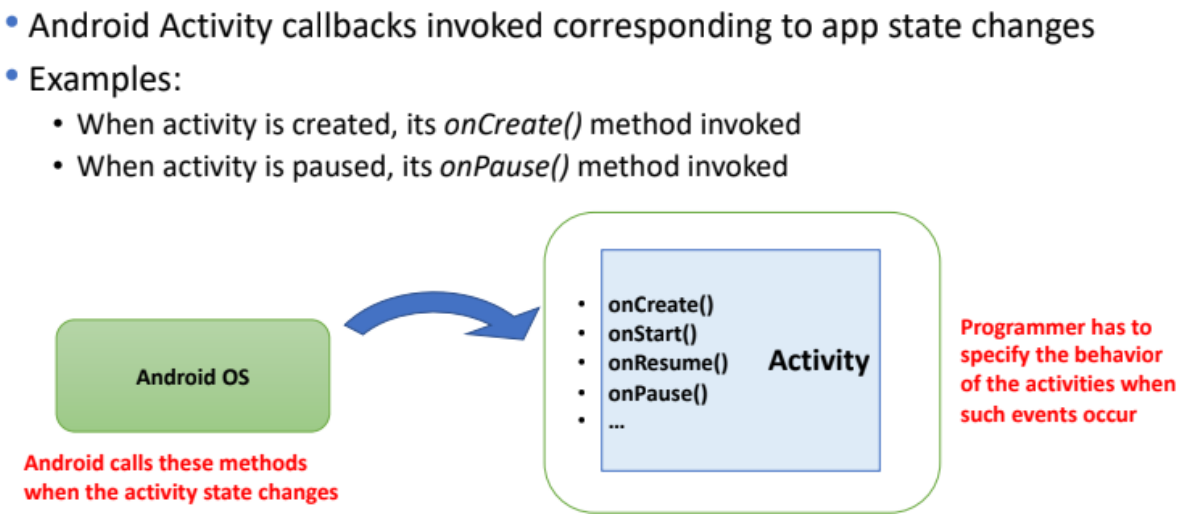
additional functionalities. They are encapsulated within the Google Services software module.
Google Services (in Google cloud):

- Maps
- Location-based services
- Game Services
- Authorization APIs
- Play Services
- In-app Billing
- Google Cloud Messaging
- Google Analytics
- Google AdMob ads

Not open source, not part of AOSP

Content providers are the third type of components. They are a standard mechanism used to export data to other applications. This is done for example by the contact list application.
A Content Provider Abstracts shareable data, makes it accessible through methods.
Applications can access that shared data by calling methods for the relevant content provider. Apps can add new data through a content provider, and they can also share it.

The fourth type is broadcast receivers. They receive events from the application or from the operative system. This is useful because Android OS (system), or applications, periodically broadcasts events.
Any app can create a broadcast receiver to listen for broadcasts, or initiate broadcasts.
Broadcast receivers typically do not interact with the UI, and create a status bar notification to alert the user when a broadcast event occurs. Then, the notification will remain in the status bar until the user decides they want to interact with the application. The notification will then launch the activity.



Methods that can be redefined:

- onCreate()
- onStart()
- onResume()
- onPause()

- onStop()
- onRestart()
- onDestroy()

Android calls all of them, a programmer can redefine all of them or a subset. onCreate() is always defined, since the app inflates layout to provide UI.

While the user is interacting, many things could happen. Well designed apps should not crash if they are interrupted, or the user switches to another app.

LECTURE 5: 15/03/2022 (Vecchio)

From the register: Lifecycle of activities. Possible states and callback methods. Some examples of state changes. Intents. Starting other activities and services. Implicit and explicit intents. Some examples: setting a calendar event, taking a picture. Broadcasted intents. Static and dynamic registration of receivers.

Activity lifecycle of an application: there are a number of methods in an activity that can be defined by the programmer, like events coming from the environment. Differently from desktop applications, mobile applications are more frequently interrupted, or the user might switch to another application.
Well designed app should not:

- crash if interrupted, or if the user switches to other apps;
- Lose the user's state/progress (e.g state of game app) if they leave your app and
- return later
- Crash or lose the user's progress when the screen rotates between landscape and
- portrait orientation (E.g. videos should continue at correct point after rotation)

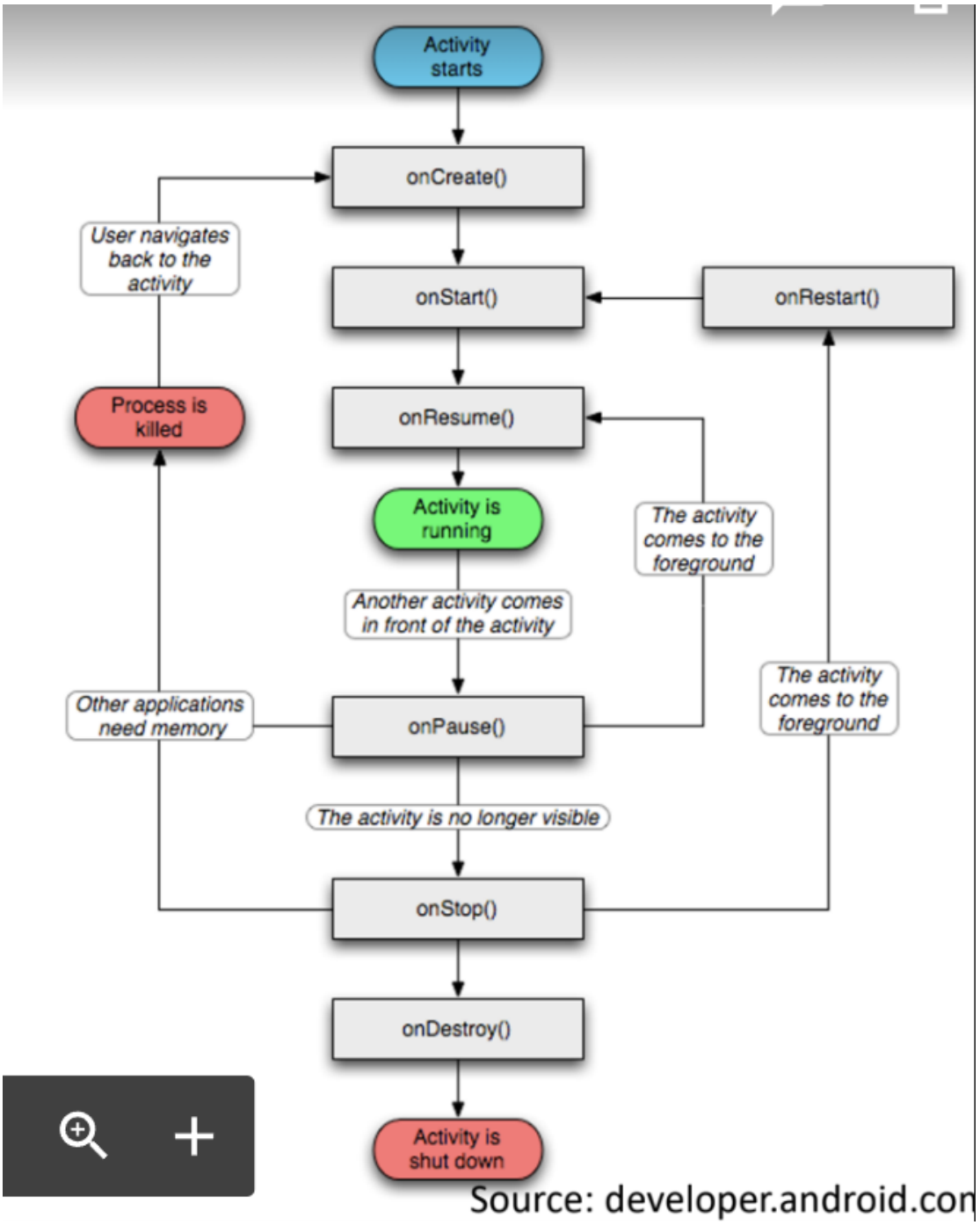
In all these cases an operating system uses some methods.

The first one: onCreate()
This initializes the activity (which is a java class) once it is created, it typically inflates widgets and places them on the screen, gets references to the widgets and sets listeners to handle user interaction.

Actually, the lifecycle of an application can be quite complex.
Let’s take, for example, a banking application. If we are using it, it is running and in the foreground. If a dialogue box comes in front of it, the banking activity goes into the paused state - app’s onPause() method is called during transition from running to paused state (like stopping energy hungry or unnecessary activities).
When we go back to the banking app, the app’s onResume() method is called to restart everything just like before.
An app is stopped if it is no longer visible nor in foreground. In this case, the onStop() method is called during transition from paused to stopped.

An activity might be stopped when the user receives a phone call, or if they start another app.
The activity instance and variables of stopped apps are retained, but no code is being executed by the activity. The onStop() method, if implemented correctly, should save progress to enable seamless restart later, release all resources, and save information to guarantee persistence.
A stopped application can go back into a running state if it becomes visible and in foreground. The app's onStart() and onResume() methods are called to transition from stopped to running state.

When an app is launched, onCreate is also called before the other two methods quoted above.
Here we have the complete lifecycle of an application.



When a device is rotated, the action kills the current activity and creates a new instance of the same activity in landscape mode. That is because the rotation changes the device configuration (screen orientation, density, size, dock mode, language etc)
Apps can specify different resources to be used for the different device configurations, like XML layout files or different images.
So, when we rotate the activity is destroyed and recreated, and new resources are loaded (as we already saw, there can be different XML layout files).

There are some situations where the onDestroy() method is executed, just before the app is destroyed. This can be implemented to free some resources like terminating additional threads created by the application.

Problem: when an activity is destroyed, and created again, some state may be lost - there are some widgets that automatically save their state and restore it when the activity is recreated. In some cases the programmer has to implement his own status saving strategies.
Before an activity is destroyed, the operative system calls onSaveInstanceState() - which saves the state required to recreate the activity later.
a Bundle object is used as a container of information to be saved, and we can use it to put some key value pairs and give it to the system when the activity is going to be destroyed.
When an activity recreates, the saved data is then sent to the methods: onCreate() and onRestoreInstanceState().
Either method can be used to restore app state data, for example using the Bundle object created before, as an argument of the onCreate() method.

Here we have some examples of common scenarios that can happen.

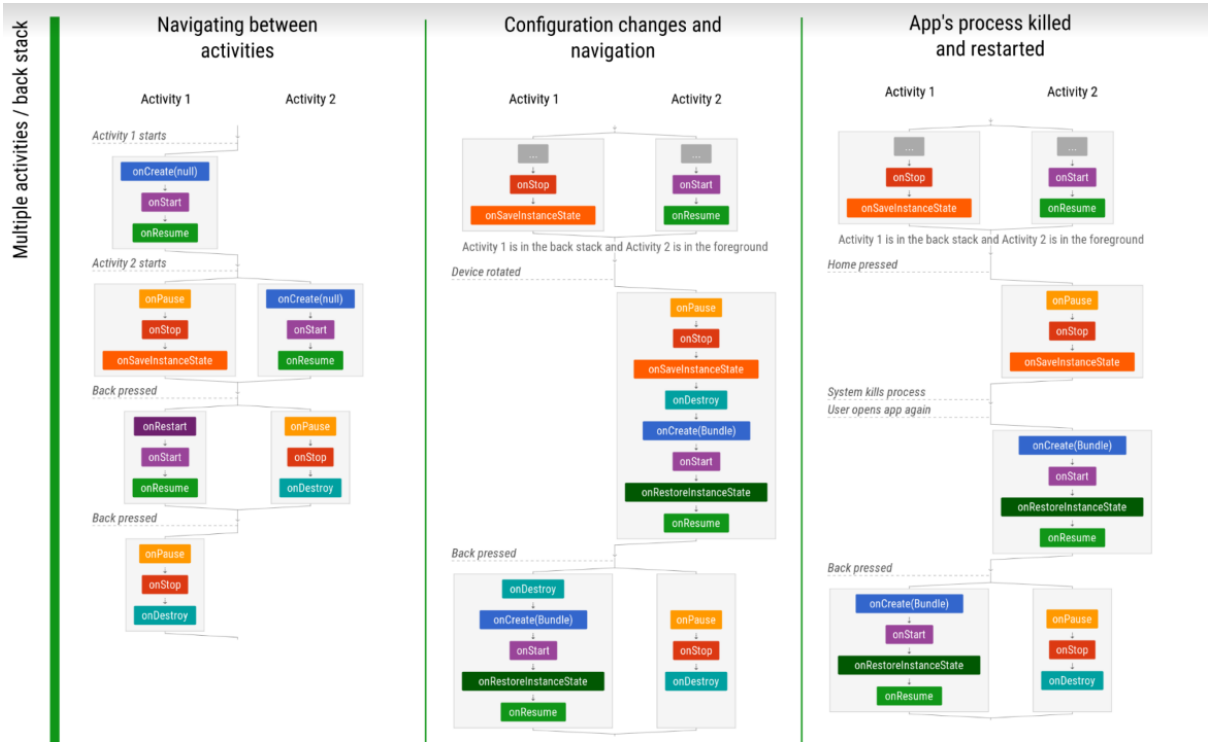
We can see the sequence of methods that are executed by Android when a situation happens.

When an app is started the methods used are: onCreate(null), onStart, onResume.
When an app is finished the methods used are: onPause, onStop, onDestroy
When the app is restarted the methods used are the same from when it was first started.

What if we press home after starting an app?
When we press home the methods used are onPause, onStop, onSaveInstanceState.
If we restart the app, the methods will be onRestart, onStart, onResume.

If the device is rotated, the methods called will be onPause, onStop, onSaveInstanceState, onDestroy, and then onCreate(bundle), onStart, onRestoreInstanceState, onResume.

Activities are organized on a stack - the first one can start the second one, that can start the third one, and newer activities are higher on the stack. The stack is LIFO.



Here we have other examples of typical function calls with multiple applications.

Intents are messages used by a component to request actions from other apps or components (for example, i have an application that must open google maps, or the camera). There are three main use cases for intents:

1. activity A starts activity B, no result back:
    a. call startActivity(), pass an intent;
    b. intent has information about an activity to start, plus any necessary data.
2. activity A starts Activity B, gets result back:
    a. Call startActivityForResult(), pass an Intent;
    b. Separate Intent received in Activity A's onActivityResult() callback;
3. activity A starts a service:
    a. activity A starts service to download big file in the background
    b. activity A calls startService(), passes an Intent
    c. intent contains info about service to start, plus any necessary data

To remind, a service is a component executed in the background, with no additional thread dedicated to it. So, there is a main thread that executes all the methods that we already know in a single execution flow - just like services. So, if we want to execute long running operations, we need to make our own thread. "Application not responding" is an error that can appear when a service is taking too much time on a thread.

Intents can be either implicit or explicit.
Explicit intents happen when components sending and receiving Intent are in the same app, whilst if components sending and receiving Intent are in different apps intents are implicit.

The intent object describes the action that we want to make.

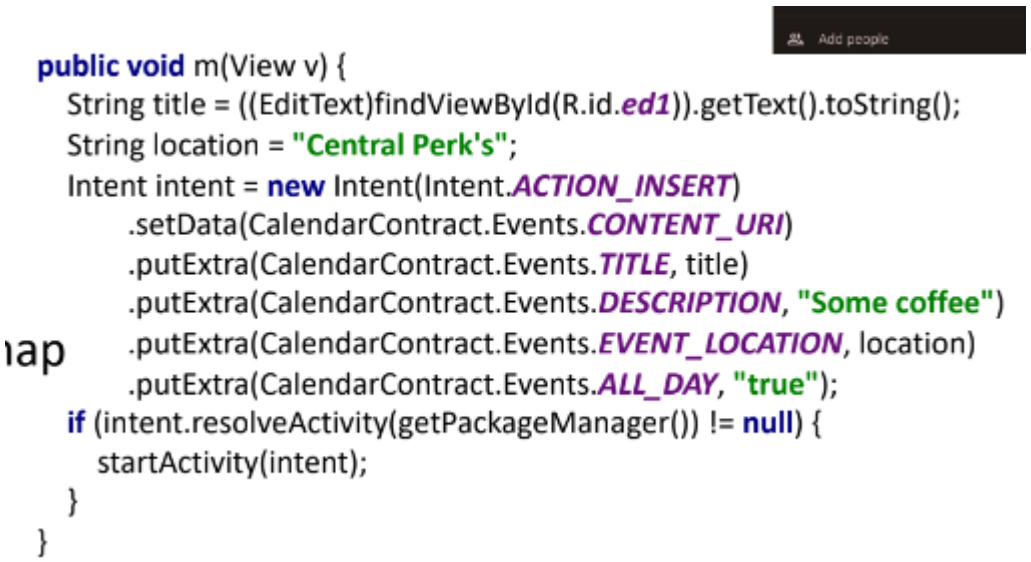
(here there is an example i did not take notes on)

Intents can pass on extras to the Activity that is going to be started. Extras are arbitrary data that can be included with an intent to pass through activities, they can be added through the putExtra() method and they can be of different types of simple values. On the receiver side they can be retrieved using a different range of methods in the format getTypeExtra(). The same approach can be followed to provide a return value to the starting activity.

if the activity that has to be started is in another app, an implicit intent has to be used: the implicit intent does not name the component to start, but specifies the **action** to be done and the **data** to perform operation on.
The system decides the component to receive intent based on action, data, category.

Actions mainly come from a set of predetermined actions. The category can be customized but it is not very useful.

When many components can carry a specific action, then android shows a Chooser.



So, once the intent object has been created and customized, we just have to provide the intent object as an argument to the startActivity method.

Some examples of implicit intents include:

- Setting an alarm
- Add a calendar event (like in the picture up here)
- Take a picture
- View or edit a contact
- Send an email
- Show a location on the map
- Start a phone call
- Open settings

For example, how do we take pictures on an app by using the Android camera app?

These are the main 4 steps:
• Request the camera feature
• Take a Photo with the Camera App
• Get the Thumbnail
• Save the Full-size Photo

To request the camera features, we must be sure that the device has a camera, first. You can allow for only devices with a camera to find your app while searching Google Play Store by making the following declaration in AndroidManifest.xml

```
<manifest ... >
    <uses-feature android:name="android.hardware.camera"
                  android:required="true" />
    ...
</manifest>
```

To take a picture with the camera app, our app needs to send an implicit intent requesting for a picture to be taken.
Since we need to have some results back, we can call startActivityForResult(), and we will then provide an implementation for the onActivityResult() method that will give us the results when available.
Also, we must check that at least one Activity can handle request to take picture using resolveActivity()
To save the full-sized photo we need to provide the camera application with a path.
Android systems have both app specific storage (data stored here is available by only your app) and shared storage (content can be available to other apps).

To save to public external storage the application needs to get the permission from the user, and must use the getExternalStoragePublicDirectory() method. Private storage needs getExternalFilesDir()

Some extra steps are required: in the androidManifest we need to add a FileProvider and a declaration of the permission, and we must define properly the path where the image is stored.

There are also a set of intents generated by the OS and broadcasted to all the interested applications.
A broadcast receiver is activated when something is received, and this method is executed by an OS thread: if long running operation you have to start a service and have a separate thread.

To receive events, a BroadcastReceiver must be registered.

- Static: in the manifest file, receives events even if app is not running
- Dynamic: by means of Java code, e.g. in Activities

These events require permissions, which will be requested to the user when the app is executed. They can also be requested through Java code. In Android 6.0+, permissions must be requested at runtime (not installation time).

The number of system broadcasts has been reduced in Android 7+ for improved energy and memory efficiency.

BroadcastReceiver: dynamic registration/unregistration

- The same receiver can be registered and unregistered dynamically

```
private static String TAG = "BroadcastReceiverExample";
MyReceiver mr = new MyReceiver();

@Override
protected void onResume() {
    super.onResume();
    IntentFilter filter = new IntentFilter();
    filter.addAction(Telephony.Sms.Intents.SMS_RECEIVED_ACTION);
    registerReceiver(mr, filter);
    Log.i(TAG, "Registering receiver");
}

@Override
protected void onPause() {
    super.onPause();
    unregisterReceiver(mr);
    Log.i(TAG, "Unregistering receiver");
}
```

LECTURE 6: 16/03/2022 (Avvenuti)

From the register: Wireless Localization: Range-based localization techniques: range estimation (TDoA,AoA). Range-free localization techniques: Centroid, APIT, DV-HOP. Discussion.

We were talking about localisation techniques. We should avoid using GPS because we don't always have the right conditions and we do not want to spend a lot of energy, especially on local nodes or sensors. That's why we look for alternative techniques. Last time we talked about range based techniques (assuming that we have anchors which communicate and send messages containing their positions, and assuming that target nodes can receive that communication, after a while target nodes are ready to compute their position).

Here is a comparison between RSS based techniques.

Technique	Advantages	Limitations
Analytical-mapping models	Simple to implement Useful for simulators design	Parameters are environment-dependent Coarse accuracy
Empirical-mapping models	Can achieve high accuracy level	Need extensive environment profiling High off-line computation overhead Poor scalability Unreliable if the environment is continually changing

Time Difference of Arrival can let us calculate the time difference instead of the difference between two timestamps, and estimate it using the speed of sound. It is energy consuming and requires extra hardware.

We have a third approach, based on the angle of arrival. In this case we need to receive a message and to know the angle at which we receive the message. In order to do this, we need special hardware, directional antennae or an array of antennae mounted on the nodes. No time synchronization is needed, but the estimated relative angles between neighbors are calculated and location is then derived from the intersection of several pairs of angle direction lines.
So, assuming that we can mount that extra hardware on our nodes, the methods rely on the angle of arrival, which can be estimated by using the index of the antennae of the specific message (if we have many antennas pointed on many different directions, at the receiving node we switch on the right one in the array and that gives us the receiving angle, which we can employ in geometric methods to, for example, draw triangles, or use trigonometry).
It is not easy to see such a solution, because it requires additional hardware and is very impractical when we have a large sensor network.

This is enough to have an overview of the possibilities that we have when we want to use gps free anchor based solutions. The major limitations (and assumptions that are not always true and raise concerns) that we have with such techniques are:

- The fact that the radio range is not always perfectly circular because of the obstacles introduced by the environment, and this adds an error to formulas;
- If i am able to receive a beacon from an anchor i should assume that the anchor is able to send a message back, but this is not always the case;
- we might need additional hardware;
- we need lack of obstructions;
- we need a clear line of sight;
- no multipath and flat terrain, which introduce noise and problems.

The idea to counter this phenomenon is to avoid estimating the distance as the first phase, and rely also on other information that generally depends on the context of the target node. Range free localization methods do not rely on distance or angle estimation in localization. They rather use proximity or connectivity information to devise the location of the target, and nodes never try to estimate their absolute distance from anchors.

There are two main approaches in range free methods.
The first ones are area-based approaches, like centroid localization and APIT (approximate point in triangle test). The second ones are multihop localization approaches like DV-HOP.
The main difference among those two approaches is a physical one. In area based approaches we must assume that all the targets are included in the range of the anchors, so we need a high density and large ranges (we only rely on one hop communication, so messages travel directly from the anchor to the receiving target node and we do not forward messages).
Multihop approaches are used when we cannot rely on a high density of nodes. Target nodes are also expected to contribute generally in the communication protocol by forwarding messages to neighbors, so collaborating both computationally and energy-wise.

Advantages: the hardware is cheap and there is no additional hardware required; the computational power is also quite low. The main disadvantage is a lack of accuracy, since we are only estimating.

The first method we will see is the Centroid Method. It is an area based approach, which means that the target node, after a while, might be able to draw a polygon whose vertices are the nodes that are sending information to the specific node. It doesn't matter the number of anchors, but the higher the number of anchors, the more precise our estimate of the location.
After that, we estimate the position of our node as the center of gravity of the polygon, computed locally like in the following formula:

$$(X_G, Y_G) = \left(\frac{\sum_{i=1}^{n} (X_i)}{n}, \frac{\sum_{i=1}^{n} (Y_i)}{n} \right)$$

In this approach, the beacon message contains the coordinates of the transmitting anchors, which are sent periodically. The application makes it so that our nodes receive at least three beacons from anchors. Of course, there are different parameters to set the periods correctly, as well as the number of beacons that are needed to estimate a position. Of course, we have an accuracy error which is higher than what we can expect using a range based method.
The main advantages are the fact that there is little overhead (computations are simple and require few beacons) and the whole system is simple to implement.
Some drawbacks: small accuracy, and the fact that we need to have a huge overlap between the radio range of all the anchors. Of course, there is a tradeoff between density and radio range of the entire environment. If I cannot install many anchors, their range must be huge.

The second technique we will see is called APIT, as in Approximate Point-In-Triangulation. The idea is similar to before. We estimate our position as the center of gravity of some triangles. In general, if we have a good coverage, we can receive many beacons and this approach and we can draw more triangles. The idea is to draw many triangles and figure out which triangles are the ones we are inside of, and then build the intersection of all the triangles - that is where the target node resides.

The tricky part is to understand whether the node is or isn't inside a triangle. The rest very much looks like the former method.
So, we will have a number of anchors that periodically send messages with coordinates. After a while every node has received a number of messages from anchors.
One of the theoretical bases behind this method is the Point In Triangulation Test (PIT), which helps us to determine whether a node N with unknown position is inside or outside of a triangle.

How does this work?
If a node is inside a triangle and it is moving, it always gets closer to at least one of the vertices. If a node is outside and it moves, there exists a direction in which it will get further from all three vertices.
So, the perfect PIT test is a test in which we assume that the node can move in any direction. We let it move and see what happens in relation to the anchors.

The perfect PIT might be unfeasible in practice, either because nodes do not move and do not have the ability to recognize the direction without moving, and it might also not be possible to perform an exhaustive test covering all the possible directions in which the target may move to.

There is a solution that proposes an approximation of this test. The idea is to use neighborhood information, exchanged via beaconing, to emulate the movement of the single node.

So, what happens is that the target node asks its neighbors for their distances to the three corner anchors. The target then compares its distance to these three corner anchors against those of its neighbors: if none of them is further from (or closer to) all three anchors simultaneously, the target assumes itself as being inside of the triangle, and vice versa.
What I need is the relative distance between the node and the anchors.
If I only need a relative distance, I can estimate it by using the RSS, which decreases as the receiver moves away from the transmitter. So, the further away a node is from the anchor, the weaker the received signal strength will be. So if my neighbor receives a stronger signal than me from the same anchor, the neighbor is closer to it than me.
In order to understand how an approximate PIT test works, we should draw circles around the three anchors on the vertices that intersect on the node. In order for a node to understand that it is outside the triangle, there should be at least one node that is not in the circles (or one that is in all three circles) so that the node sees that at least one neighbor is further or closer to all the anchors.
What if the node is inside? We still draw the circles, and we look at all the nodes. No node should be able to be inside all circles, or outside all three.
This is how each node works:

Receive location beacons (X_i, Y_i) from N anchors;
$InsideSet = \emptyset$;
for each triangle $T_i \in \binom{N}{3}$ triangles **do**
  if Point-In-Triangle-Test (T_i) == TRUE **then**
    $InsideSet = InsideSet \bigcup T_i$;
  end if
end for
Estimated Position = CenterOfGravity($\bigcap T_i \in InsideSet$);

In APIT, there are several error scenarios. It depends on the relative position of neighbors. For example, In-to-out error happens when a node is inside and thinks that it is outside, because it has some neighbors that are outside. Out-to-in error happens when the node is further from the anchors than its neighbors.

Experimental results show, though, that the error percentage decreases as the node density increases. In particular, Out-to-in error percentage lowers whilst In-to-out error percentage grows but staying under 2%.

We can represent the maximum area in which a node will likely reside by using a grid SCAN algorithm. We position all the triangles on the frid and, for each square in the grid we put a number, starting from 0 at the beginning.
If the node is inside a region, the corresponding grid squares are incremented. If the node is outside, the grid regions are decremented. Of course we have to deal with an error.

So, to conclude this, the advantages of APIT are a small overhead, and more accurate results than the centroid method. The main drawback is the fact that we have problems determining the position of a node located out of all anchor triangles.

We have another type of range free localization methods, which are called distance propagation methods. It is not always possible that a target trying to locate itself is in communication range with at least three anchor nodes, because of the limitation of the transmission power; mechanisms for multihop localization have been proposed to extend the localization process over a larger geographical extent.

The technique we know, called DV-HOP, does not estimate the absolute distance but obtains a logic distance estimation starting from the number of hops. We will then convert this logical distance into a physical one, so one that can be measured in meters.

How does this work?

Again, we will have anchors which independently broadcast their beacon message which embeds a location and a hop counter, which is initially set to 1 and increased at every hop. This way, the beacons will flood the network and each target node will receive beacons from further anchors. Then, each target node will identify the shortest path to each anchor node and try to estimate its distance.

The idea is to compute the number of hops between any two anchors (Ai, A j) and estimate the average 1-hop distance by dividing the sum of physical distances by the sum of logical distances.

This length depends on the type of network, radio channel, radio range etc… Also, we are not in a theoretical scenario, so we may have long distances or shorter distances depending on the number of obstacles in the way. So, we need to compute the distances locally to have less errors.

So, we have 2 phases. In the node update phase, the anchor nodes broadcast beacons with coordinates and numbers of hops. When the targets receive the beacons, they set up tables locally where they put all the information that they receive.
In more detail: When a target N receives a beacon from an anchor, it maintains the record (X , Y , h) for each anchor A , where (X , Y) represents the location of the anchor, and h , the number of hops from N to that anchor A. Then, N increments h and forwards the beacon

to neighbors in its radio range. At the end of this step, targets know about the locations of anchor nodes and the hop counts to reach them.

Anchors can maintain the same tables! While target nodes do not know their position, anchors do and can calculate the euclidean distance between all the nodes.
So, anchors themselves can calculate the 1 hop distance.

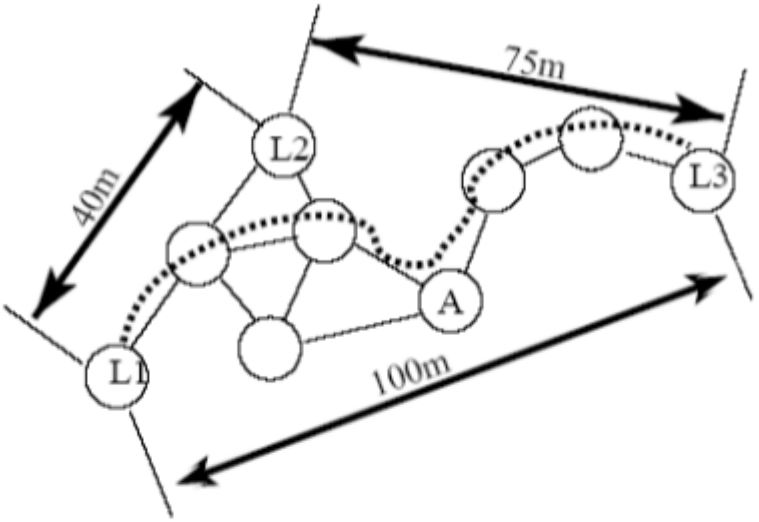
The next phase is called 1-hop distance estimation phase, and it takes place when an anchor node receives the locations and hop counts from other anchors. It can calculate the estimate average hop distance, referred to as correction factor c_i

$$c_i = \frac{\sum \left(\sqrt{(x_i - x_j)^2 + (y_i - y_j)^2} \right)}{\sum (h_i)}$$

where i is the anchor that calculates correction, j are anchors known to i, x *and* y are anchor coordinates, h is the hop count from anchors

After this phase, these factors will be broadcasted between anchors. A node that receives this correction, will forward it and then stop forwarding corrections.

Target node localization: each target uses the correction received from the closest anchor as the estimated 1-hop distance. It then multiplies the 1-hop distance by the hop counts to other anchors to estimate its physical distances to them. After getting distance estimates to at least three anchors, a target can use trilateration to approximate its location.



L_1 computes the correction $\frac{100+40}{6+2} = 17.5$.
L_2 computes a correction of $\frac{40+75}{2+5} = 16.42$
L_3 a correction of $\frac{75+100}{6+5} = 15.90$.
A gets its correction from L_2

Distance to L1 = 3 x 16.42, to L2 = 2 x 16.42, to L3 = 3 x 16.42

This method is very simple. On the other hand, there is a large overhead, and there might be many estimation errors which depend on the number of anchors that a node can hear. Also, it only works for a network that is uniform in all directions (isotropic), otherwise the 1-hop distance will not be accurate as connectivity will be less correlated with range.

In conclusion, localization is important in CPS and WSN. Many proposals presented to address localization issues :Range-Based localization techniques are accurate but costly. Range-Free localization techniques are cheap but inaccurate.

LECTURE 7: 22/03/2022 (Vecchio)

From the register: Context-aware computing. Definitions of context. Context sensing. Designing and building context-aware applications. Issues when building context-aware applications. Activity recognition: a step counting application.

part on the projects and paper presentations which i did not transcribe

Now, let's talk about context aware computing. Humans use information coming from the context when they talk with each other. Information coming from the environment is usually not available when interacting with computing.

The idea behind context-aware computing is to include some information coming from the context as a sort of implicit input that can be used to improve the interaction between humans and machines. It is a step towards ubiquitous computing, “calm” computing, where computing is not always at the center of the user’s attention but moves to the side when not explicitly needed.
Applications that use context, whether on a desktop or in a mobile or ubiquitous computing environment, are called context-aware. They have become quite popular in the last twenty years, for two reasons.
The first one is the fact that sensors have become much cheaper; the second one is that users are much more “mobile” than before. If I move to different locations, all interactions are different. We need to have a means for the services to adapt appropriately, in order to best support the human–computer and human– environment interactions.

Context aware applications can be found in many domains, like wearable or mobile computing, adaptive and intelligent user interfaces, intelligent environments and augmented reality.

But… what is context? We have a general and maybe vague idea of what context is.

Here is a list of definitions:

- Schilit and Theimer (1994): refer to context as location, identities of nearby people and objects, and changes to those objects
- Brown et al. (1997): define context as location, identities of the people around the user, the time of day, season, temperature, etc
- Ryan et al. (1998): define context as the user’s location, environment, identity, and time
- Dey (1998): enumerated context as the user’s emotional state, focus of attention, location and orientation, date and time, and objects and people in the user’s environment

These definitions progressively add elements to the idea of context, but they all define context by examples and they are difficult to apply.
Some would consider either context to be the user’s environment, or even consider it to be the application’s environment.

A more recent, more general definition: “Any information that can be used to characterize the situation of an entity. An entity is a person, place, or object that is considered relevant to the interaction between a user and an application, including the user and application themselves.”
Dey and Abowd (2000a)

In this definition there is the adjective, “relevant”. Let’s suppose that we want to develop a restaurant booking application. There is some information that can be implicitly collected, like humidity, number of people in the nearby area, position. All this information is implicit, but just the position is relevant!

Context information is generally obtained by sensing the environment of the user. Environmental sensing is relatively easy, but sensing humans is quite harder.

You need to follow the five W + one H:

- When: time (this is the easiest one to collect)
- Where: location (we might use GPS but in some cases we need the logical location of the user, like “in which room are they” or “in which aisle of the supermarket”)
- Who: identification (by face, id, movement)
- What: the action that they are doing, like running, eating etc. (in some cases it is not really different, like when the user is running… not as easy if you are trying to understand if they are cooking, for example)
- Why: the reason behind the actions (extremely hard to sense)
- How: their mood while doing something (you can use gesture recognition and cameras)

Pervasive computing integrates location sensing, user identification, emotion sensing, gesture recognition, activity sensing, user intent.

In practice, there are some types of contexts that are more important than others. These are location (where), identity (who), time (when) and activity (what).
These pieces of context can help us point to other contextual information of the user: for example, given a person’s identity, we can acquire phone numbers, addresses, email addresses, birth date, list of friends, relationships to other people in the environment, etc; given an entity’s location, we can determine what other objects or people are near the entity and what activity is occurring near the entity.

From another point of view, context can be organized in a set of physical properties, measured by hardware sensors, and other logical properties, that can be inferred from the recorded physical context or from the user’s interaction with the application.

For example, the actions we could take might be different if we only knew that the user was texting, versus if we knew that the user was also driving whilst texting.

There are three main categories of possible usages of context information:

1. Presentation of different information and services to the user (for example, a list of printers in the nearby can be updated as the user moves in a building)
2. Automatic execution of a service, (for example, a remind that can be triggered as the user passes in the nearby of an interesting spot for whatever reasons, or an automated turn on of silent mode during class lectures)
3. Tagging of context to information for later retrieval (for example, information of humitìdity and temperature during a running session)

The design of a context-aware application must endure different steps.

- Specification: what context-aware behaviors your application will have and in which situations each behavior should be executed.
    - (example, we want to light up a LED depending on the activity level in a remote room)
- Acquisition:
    - determining HW and/or SW sensors required to acquire the context identified
    - using the API to communicate with the sensor and define how to be notified when changes occur

    - store the context, combine it with other context
        - (example, we write software to analyze frame-to-frame changes in the cctv image of that room)
- Delivery: specifying how such context information should be delivered from the sensors to the (possibly remote) applications that will use the context
    - (example: we make the percentage of change in activity available to all interested applications, by using a publish-subscribe approach)
- Reception: application specifies what context it is interested in (possibly indicating from which sensors) and receiving that context
    - (Example: the application wants to receive all the values that are greater than a predetermined threshold)
- Action: analyzing all the received context to determine what context-aware behavior to execute, and then execute actions
    - (Example: The application analyzes the received activity changes and lights up the led when it is needed)

To build such an application, there are three ways:

1. No support, we write everything from scratch, we read the values from the sensors and implement all the low level techniques for filtering out the noise;
2. Widget, or object based approach - sensors are components that can be used by the application;
3. Blackboard based system.

A context widget is a software component that provides applications with access to context information - we hide the complexities of sensors behind them. We want to separate the output of the sensors from the internal working mechanism, and abstract the context information to suit the expected needs of applications, and to make reusage possible.

Access to context data can be made through querying and notification mechanisms, and these mechanisms can be reused and combined. For example, if we have a sensor that can detect the presence of a person in a room, if this sensor detects two people, another sensor might be built on top of it to signal a possible meeting between two people.

We already cited blackboards, without explaining their meaning. The blackboard approach is based on the idea that components can place information into the storage system, and read/remove this information. Blackboards may have the ability to notify components when information of interest has been added to the blackboard.

Blackboard approaches are sometimes inefficient, particularly as the amount of data in the tuple space grows: in this situation, searching for specific tuples becomes harder.

When building context aware applications, we must face and discuss several issues.
First of all: context is a proxy for human intent, which is what we would like to have but is too difficult to obtain. Applications would like to use this intent to adapt appropriately, but since we can only sometimes get some circumstantial evidence about the real human intent, applications tend to sometimes fail.

One solution would be to add additional information to the context, or express a confidence level in their beliefs about their sensed information and set a threshold under which to ask for confirmation of the information.

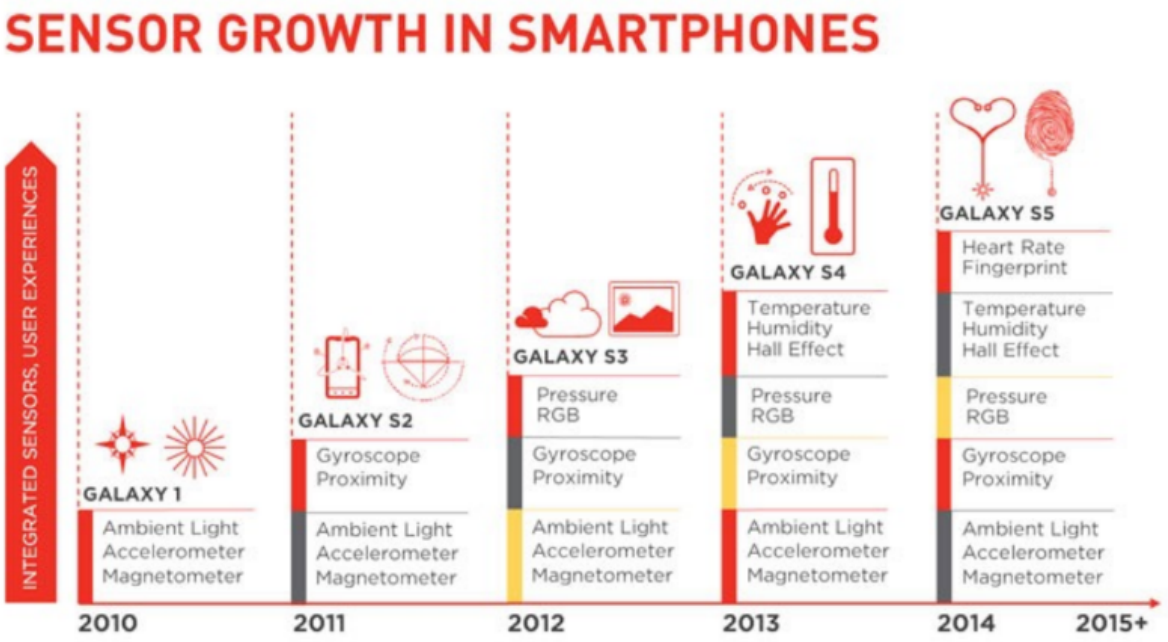
One second issue is context ambiguity. context sensors can sense incorrectly, fail, or be unsure about what they sensed; context inferencing systems can inaccurately reach conclusions about a situation.
One solution is to add multiple sensors, even with different hardware (this solution is known as sensor fusion); another possibility is to rely on past information on context, since it mostly shows some coherence over time.

Context information can be used in many different ways. Sometimes it is based on rules, “if this then that”, very easy to build and very intuitive but can create collisions between rules and it might be difficult to evaluate the system in its entirety as the rules add up.
Some other times it is based on Machine Learning techniques, where developers collect data on the user’s possible context scenario and the types of adaptation desired, to then apply ML on the collected data to learn the probabilistic relationships between the situations and adaptations. This might require a large amount of data to let the models learn, but avoids having to hardcode a lot of rules.

Another concern is privacy about the user and the user’s surroundings. We must always keep in mind this aspect, since there are always concerns about the fact that information could be disseminated to components that are not always trustworthy. Developers of context-aware applications need to ensure that a user’s privacy is maintained and that information is not being used inappropriately, or in a manner that the user feels inappropriate. The user must always know why some specific information is needed.

A final problem is that the output of the application might be not totally comprehensible to the user in some cases, because if part of the input comes from the environment the user could not fully understand how the output was obtained. This can be solved by providing appropriate feedback to users, to indicate which actions were taken or why. The problem here is that this is challenging, even for rules based systems - and even more for ML based ones.

In a few years, the number of sensors in smartphones has grown. - smartphones might use sensors to sense rotation, light, face proximity, location.

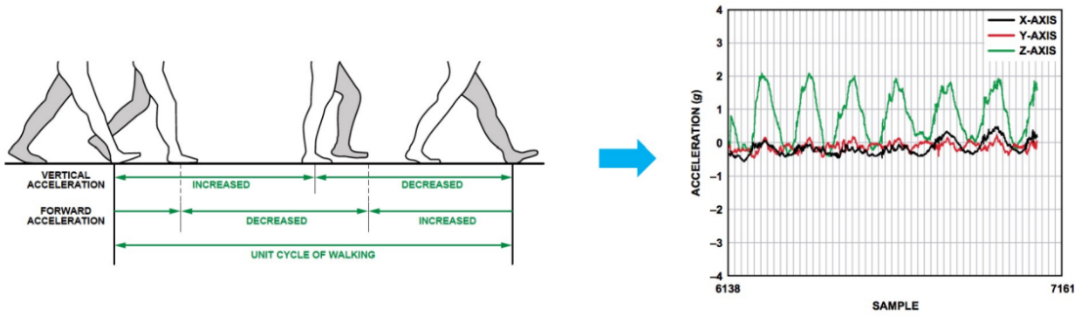


There are many benefits of using context: for example, applications can be proactive (for example, they can give automatic reminders), filter information, reduce the cognitive load of users, and let them interact with intelligent environments.

Let's now see some concepts about recognizing the activity of the users and how we can use the sensors available in an android smartphone.
As an example we will see how to build a simple step counter. Some motivation on why to develop applications that make information about their physical state available to the user: in particular, the risks associated with a sedentary lifestyle.

This is one of the reasons why a lot of devices like pedometers, fitness trackers etc. became so popular in the last few years. Tracking makes users aware of their activity levels and motivates them to exercise more.

How does a simple step counter work?
When a person moves, there are variations in acceleration on the three axes. A smartphone has three axes as well and can detect movements with sensors - the alignment of the axes, though, depends on the orientation of the device.
Vertical and forward acceleration increases/decreases during different phases of walking, and walking causes a large periodic spike in one of the accelerometer axes (which one? As we already said, it depends on orientation)



What can we do? We can select the axis that provides the larger changes, or take the module of the acceleration.

The first step in the step detection algorithm (pun not intended) is smoothing the signal. The signal is noisy because the accelerometers are cheap, because every step is kinda different and the device is mostly not firmly attached to the body. In this particular application noise is not a huge problem, but removing it is better.
Another problem is dynamic threshold detection: we would like to find a threshold such that each crossing of it signifies a step, but thresholds cannot be fixed due to the fact that magnitude depends on phone orientation and walking style. A solution could be to track min and max values observed every 50 samples, and take the average as the threshold.

So, a step is indicated by crossings of the dynamic threshold, and defined as a negative slope when the smoothed waveform crosses the aforementioned threshold.

If we implement a pedometer like this, we will encounter a lot of false steps, due to vibration caused by daily activity like public transport or general vehicle usage.
We can assume a periodicity of walking/running. For example, we could assume that people can run 4 steps per second and walk 1 or 2 steps per second. Any negative crossings that occur outside periods can be discarded.

LECTURE 8: 23/03/2022 (Vecchio)

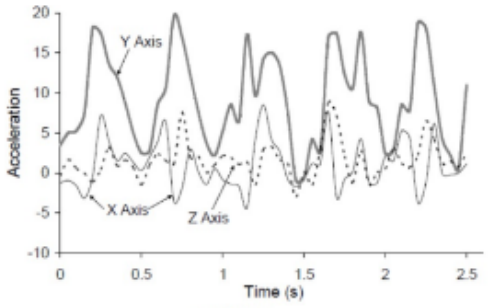
From the register: Acceleration patterns associated with different activities. Applications of activity recognition. Google API for activity recognition. Android sensors. An exercise: implementing a step counter in Android.

Let's continue what we interrupted yesterday.
The previous step detection algorithm is simple: it could use more sophisticated signal processing algorithms for smoothing, like frequency domain processing, but we only looked at it to understand the main idea behind it.

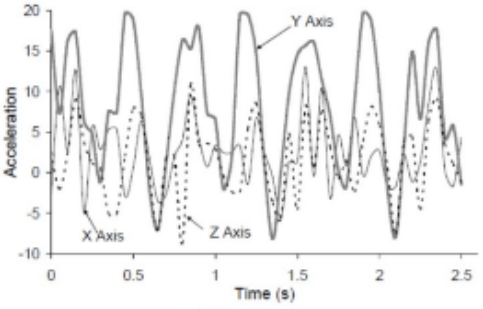
Now, let's talk a bit more generally about activity recognition.
We want our app, for example, to take step activity and categorize it into one of 6 activities. The typical solution is to rely on a machine learning classifier. We would split the data in

snippets of a few seconds and then extract the useful features from every snippet and classify them.

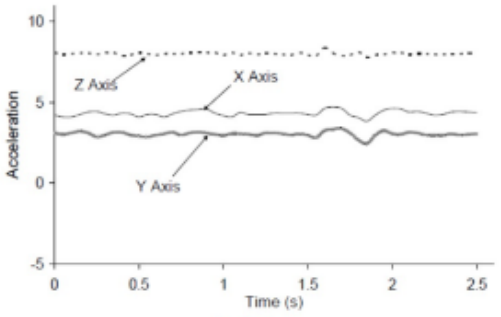
Just by looking at the signals we can see differences.



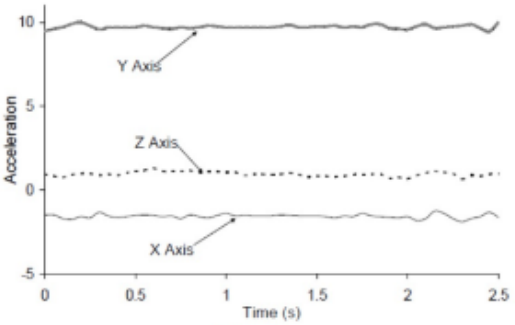
(a) Walking



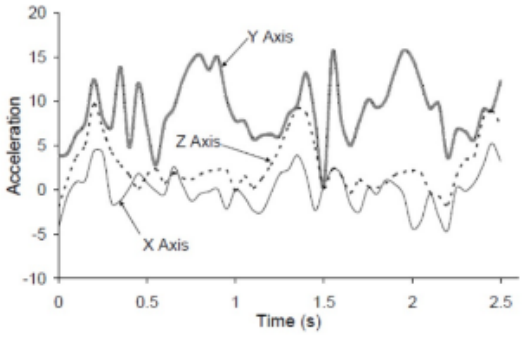
(b) Jogging



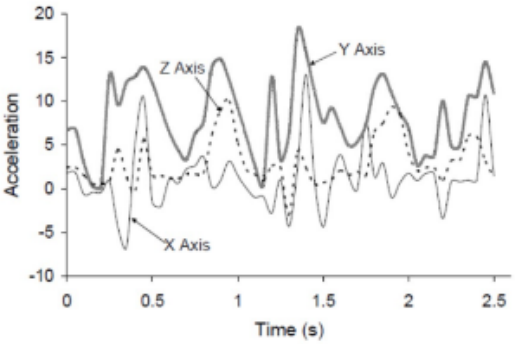
(e) Sitting



(f) Standing



(c) Ascending Stairs



(d) Descending Stairs

There are countless applications that track the activity of the users. At first they were only tracking calories, distance and physical activity type. Nowadays they also track stairs climbed, logging and context of activities and even sleep.

Some applications have health monitoring purposes, with the goal of making clinical monitoring pervasive, continuous and make so that the patient data is much more adherent to the activity of their daily life. Another application could be fall detection software, especially useful for elderly people for which fall is one of the leading causes of death.

By collecting huge amounts of this kind of data, we can widen our scope even to an international point of view, and we can look at the differences in activity and average daily steps worldwide.

Let's try to understand how to put these ideas into practice. We have Google APIs that can detect a smartphone user's current activity. It currently detects 8 states:

- In vehicle
- On bicycle
- On foot
- Running
- Walking
- Still
- Tilting
- Unknown

This can be done in two forms, either transition based (so the app receives the intents when an activity starts or stops) or periodic.

Google Play Services is a component containing closed source code that provides an interface to several APIs.

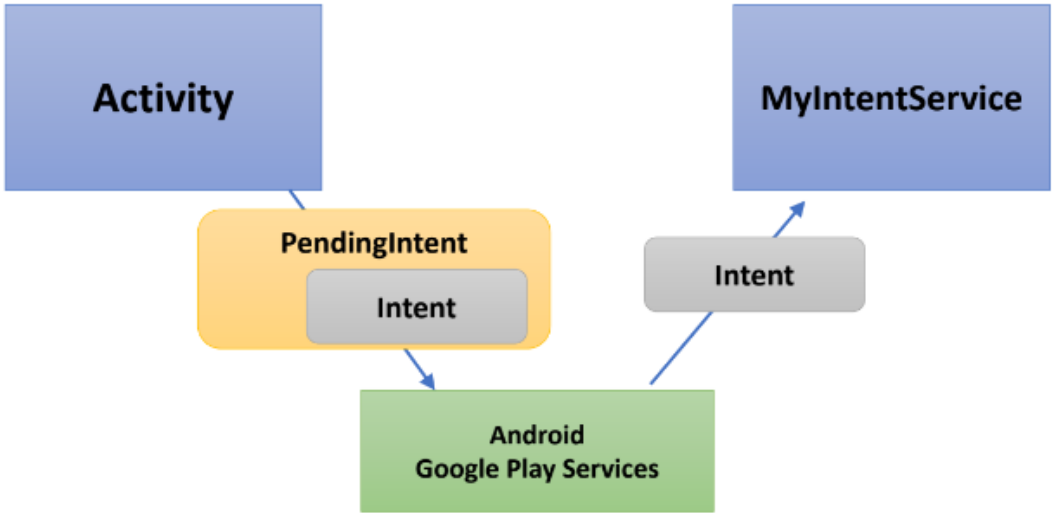
For example, if we want to log the activity the user is performing, in mainActivity, for example in onCreate(), we can put this:

```
final int PERIOD = 1000; //in ms

ActivityRecognitionClient mActivityRecognitionClient =
                            new ActivityRecognitionClient(this);

Intent i = new Intent(this, MyIntentService.class);

PendingIntent pi = PendingIntent.getService(this, 1, i, PendingIntent.FLAG_UPDATE_CURRENT);

Task<Void> task = mActivityRecognitionClient.requestActivityUpdates(PERIOD, pi);
```

Client for Google API

A container for an intent that will be fired in the future

This other picture might make the situation clearer.




```
public class MyIntentService extends IntentService {

    public MyIntentService() {
        super("MyIntentService");
    }

    @Override
    protected void onHandleIntent(Intent intent) {
        ActivityRecognitionResult result =
                                ActivityRecognitionResult.extractResult(intent);
        ArrayList<DetectedActivity> detectedActivities =
                               (ArrayList) result.getProbableActivities();
        for (DetectedActivity activity : detectedActivities) {
            String act = convertToString(activity.getType());
            Log.e("AR", "Detected activity: " + act + ", " +
                    activity.getConfidence());
        }
    }

    String convertToString(int a) {
        switch (a) {
            case DetectedActivity.ON_BICYCLE: return "on bycicle";
            case DetectedActivity.IN_VEHICLE: return "in vehicle";
            . . .
        }
    }
}
```

IntentService: a Service that carries out the requested operations in an additional thread, that is terminated automatically when there is nothing to do anymore. This is useful if a service needs to operate for a lot of time. In particular, from the point of view of the programmer, the only thing to do is provide an implementation for the onHandleIntent(intent) method, that will be executed using the additional thread whenever there is something to do.

In order to use these APIs we must add the permission in the manifest file and the dependency in the in app module build.gradle.

Google also has an Awareness API, that can be used to provide information about the activity of the user and their location - there is also information about the headphones and about beacons.

It combines existing APIs.
It gives information using two methods:

1. Snapshot API returns cached values of Location, Activity, Time, and it is less accurate but optimized for battery and power consumption;
2. Fences API is used to set conditions to trigger events, used for implementing smart reminders.

At a lower level, we can also use the sensors available on Android Smartphones. Android provides an API to access sensors and obtain data with an uniform interface. in general, applications must:

1. Obtain the list of all available sensors
2. Register listeners to one or more
3. Process the data received through callbacks

This somehow corresponds to the widget-based context support, and these sensors can be hardware or software, generally they provide abstractions.

These available sensors can be classified into environmental or motion/orientation sensors.
The real sensors are different from device to device, and their availability depends on API level.

SensorManager provides methods for obtaining the list of sensors, and registering listeners.
It is possible to restrict the list of sensors we are interested in. The list can be quite long, even more than 20. For each sensor we can get information like name, type, power needed, vendor. It's also possible for the sensor to store some of the recorded values internally in a FIFO queue, to reduce the amount of energy needed for that sensor, since the application processor can go into low power mode; in that case, the sensor can still collect information that can later be given back to the application when needed.

Once we do have an instance of a sensor, we can register a listener - which is an object that implements the SensorEventListener interface and contains a couple of methods. The first is executed when there is a new value coming from the sensor, and another one when the accuracy of the sensor changes.

- To register a listener and receive events

```
Sensor s = sm.getDefaultSensor(Sensor.TYPE_LINEAR_ACCELERATION);
sm.registerListener(this, s, SensorManager.SENSOR_DELAY_GAME);
```

Object that implements the *SensorEventListener* interface

Requested sampling rate

```
@Override
public void onSensorChanged(SensorEvent event) {
    float[] m = event.values;
    . . .
}
@Override
public void onAccuracyChanged(Sensor sensor, int accuracy) {
    Log.i(TAG, "Accuracy changed");
}
```

Here is a list of Android's sensors.

- Environment
  - Ambient temperature
  - Barometer
  - Light
  - Humidity
- Motion and orientation, which are used to detect movements such as tilt, rotation, shake, swing
  - Accelerometer, which returns acceleration (always HW), and includes gravity, so if it free falling everything is 0
  - Gyroscope, which returns angular velocity (always HW, whilst the other can also be SW) and if we want to compute the rotated angle integrates the velocity; errors in computation of the angular velocity are frequent and might be accumulated, which results in a drift in computing angles
  - Gravity, which returns direction of gravity, useful for understanding device orientation, uses a low pass filter or more complex techniques
  - Linear acceleration, which returns acceleration minus gravity
  - Rotation vector:
    - returns device orientation in the form of a set of values (azimuth, pitch and roll, which are the angles around respectively the z, x and y axes) by using a fusion of gyroscope, accelerometer and geomagnetic field
    - (the accelerometer provides information about pitch and roll because of gravity, the geomagnetic field provides information about azimuth because of the magnetic north, the gyroscope can provide information that is then fixed by the other two)
    - the orientation is then provided as a rotation vector that can be expressed in several ways; either (x, y, z) or ($x \cdot \sin(\theta/2)$, $y \cdot \sin(\theta/2)$, $z \cdot \sin(\theta/2)$)
  - Significant motion: used as a trigger to understand device is picked up
  - Step detection: an event for each step
  - Step counter: returns the cumulative number of steps since reboot

- Magnetic field: amount of geomagnetic field along 3 axes

here is an example on the slides, plus the exercises, i didn't take notes on

LECTURE 9: 29/03/2022 (Avvenuti + Vecchio)

From the register: Distributed Hash Table. The Publish-Subscribe problem, centralized vs Peer-to-peer architectures. Distributed Hash Table: properties, structure, operations, applications, API.

This is the basic mechanism that allows us to build a peer-to-peer network in an efficient way.

Publish subscribe pattern is a messaging pattern where senders of messages (or publishers) do not program the messages to be sent directly to specific receivers, called subscribers. but instead categorize the messages into classes without knowledge of which subscribers, if any, there may be.
Similarly, subscribers express interest in one or more classes and only receive messages that are of interest, without knowledge of which publishers, if any, there are.

So, when we build such a system, we need to implement at least three services. First, how are publishers advertising that they are going to publish messages? Second, where are we storing messages before they will be read by subscribers? Third, how are subscribers going to tell the system that they are willing to receive messages?

We do not have explicit receivers, so we need some filtering mechanism. We might have different kinds of publish-subscribe systems.
The first one is a topic-based system. In this system, messages are published to topics, or logical channels, Subscribers will receive all messages published on given topics to which they subscribe. It's the publisher that defines the topics.
The second one is a content-based system, where messages are only delivered to a subscriber if the attributes or content of those messages match constraints defined by the subscriber. It's the subscriber which classifies the messages.

In general, we have some nodes that act as subscribers, others as publishers, and we must choose how to make them interact with each other.
Each publisher needs to advertise a topic, and the implementation must guarantee that both the topic and the stream of messages are stored somewhere.
Subscribers want to express their will to receive messages, by sending messages to the "cloud": where do I send the message? Should I send it to a central server or to distributed nodes?
What about the delivery of messages? Once we have a publisher, a topic and the interested subscribers, I need to ensure the flow of messages to them.

If we use a central server, it's easy. Publishers send messages and topics to the server, the subscribers send messages to the server when they want to receive messages. There are, though, several drawbacks: there is a single point of failure, of course, and there might be a

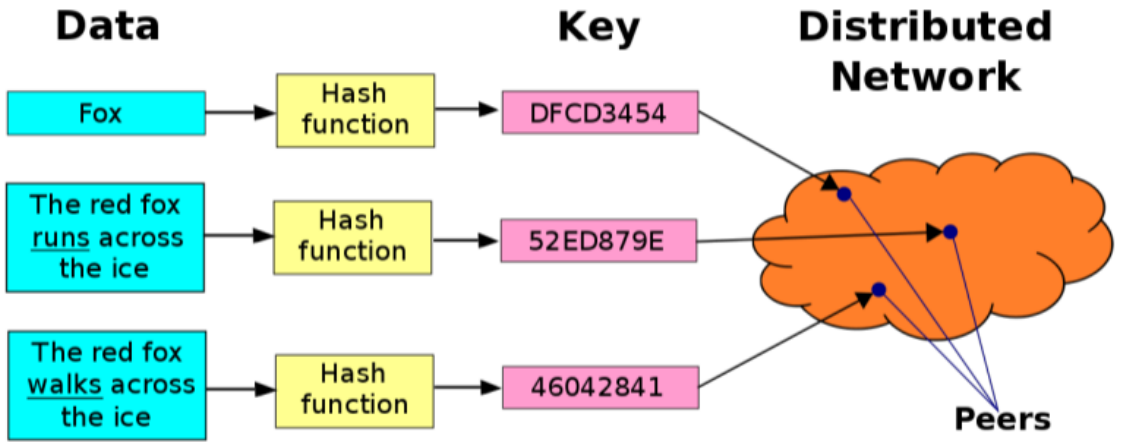
lot of traffic and computation in the server. Also, this system does not scale, like any centralized solution.

On the other side, we can use a peer to peer architecture. The nodes behave in a symmetric way, and this function can change overtime dynamically. There is no centralized control, but a large number of heterogeneous and unreliable nodes.
Basically, when we want to use a peer to peer method for storing information and retrieving it, we face the lookup problem. We must be able to know where a resource is. So, in terms of publisher subscriber, we need to introduce repositories, which are nodes whose role is to associate resources with keys, so that resources can be sought.
When a publisher wants to advertise a topic and write messages, it must have a repository to store the data. The repository must also be known by the subscriber, which must know how to reach him.

Again, we are tempted to use a centralized solution: another device called “coordinator” - we may have more than one thereby distributing the information.
Publisher sends messages to a given coordinator. When the coordinator receives the messages, it knows in which repository to put the information. The subscriber will ask the coordinator for the messages, and the coordinator will alert the repository. Once the repository knows about a subscriber, it will autonomously update the subscriber.
There is still a single point of failure - the coordinator. Another problem, the complexity: there will be O(n) messages on the coordinator, which will not store messages but still have a complexity given by maintaining the status.

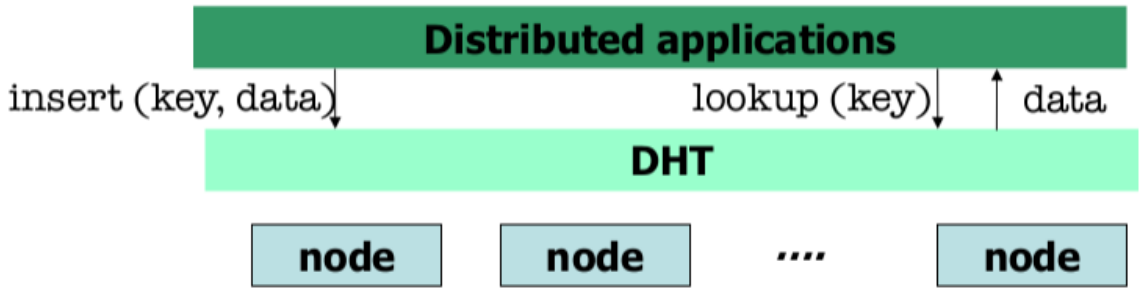
If we want to take advantage of the peer to peer network, we can simply flood the whole network of messages.
For example, when a publisher starts producing messages, it will send them to all the nodes. The subscriber will do it as well. Of course, there might be a problem: the network has too much traffic. So, there are O(n) messages per lookup.

So, what can we do? We can introduce a new idea, a way to store data when this data is associated with a key, and use this key to find where data is stored.
So, we use a distributed hash table.
It is a distributed system which mainly provides a lookup service, which means that when a node wants to locate data, it must provide the associated key - the table must store keys and values. Of course, lookup means that any node must efficiently be able to retrieve the value associated with a given key.
Keys are unique identifiers, and the values can be an address or any other kind of data. The mapping from key to value must be distributed (we need a rule). Responsibility for maintaining the mapping from keys to values is distributed among the nodes, in such a way that a change in the set of participants causes a minimal amount of disruption. The main advantage of a DHT is that nodes can be added/removed with minimum work around re-distributing keys.



The basic structure of a DHT is composed by:

- An abstract keyspace
- A keyspace partitioning rule to split between nodes
- An overlay network to connect the nodes and allow them to find the owner of any given key in the keyspace.



The DHT API: the DHT must provide at least the insert function - which will guarantee that the key will be stored in one of the participating nodes (nodes behave as hash buckets) and the lookup function, to look up the data in the nodes. The pairs of key and data must be distributed evenly and in a balanced way among nodes.

End of the Avvenuti part of the lecture, beginning of the Vecchio part where he mostly presents last year's projects and i didn't take notes

Mobility as a service: the idea is to provide the user with mobility services, by using shared transportation means. The concept describes a shift away from personally-owned modes of transportation and towards mobility provided as a service. This is enabled by combining services provided by private and public transportation providers through a unified gateway that creates and manages the trip, which users can pay for with a single account.
There are different subtopics, like on-demand mobility solutions; unified gateway that creates and manages the trip; journey planner; booking; single account; carpool; ridesharing; self driving cars; contactless payment.

LECTURE 10: 30/03/2022 (Seminar)

From the register: Smart wearable systems for continuous health monitoring: systems for personalised healthcare, application fields. Monitoring of human activities: approaches, wearable sensors. A case study: frailty status assessment through gait analysis and machine learning (guest lecturer: Domenico Minici).

Smart wearable systems for continuous health monitoring.
We are going to talk about how in research: you start from the problem, by knowing the field. You move step by step and try to find a solution. We will start from an introduction of the problem, then we will say something about sensors, and then we will talk about a solution.

Healthcare is a system in which different entities interact. The current situation: we have general practitioners which we refer to when we have symptoms. The doctor behaves like an access point to the healthcare system and can refer us to other actors.
It is a reactive approach to disease. Examinations are snapshot-like, and patient data is not always handled in the best way.
There are also many financial challenges: expanding and aging populations have a higher incidence of many diseases, some of which are chronic and long term. Medical technology investments and infrastructure is costly for the system, and there are staff shortages, plus a growing demand for a larger ecosystem of services. There is also the fact that most care should be moved from the clinic and hospital to the home. One of the most expensive and crowded fields is geriatrics.

We do not want to replace doctors and nurses. The technology should only make their job easier, by moving testing from the clinics to the domestic environment, to prevent conditions in the future and allocate time and resources better in the hospitals. Another reason why technology and a change of approach could be useful is in order to better manage and share a patient's data, and generally increase the availability of healthcare.

We can summarize this new approach with the P-health term, that means: Preventive, Pervasive, Personalized, Pervasive, Participatory, Patient-centered.

Subjects expect care to be available when it is most convenient and safe for them, and the use of artificial intelligence and wearable technology can help us by providing virtual care, at-home prescription delivery, remote monitoring, digital diagnostics, decision support and social support.

There are many application fields for wearable technology:

- Fitness and well-being: improve fitness experience and monitor overall well-being (e.g., fall detection)
- Hospital patients: better monitoring of patients before and after interventions (longitudinal study)
- Chronic disease patients: better understanding on the disease through continuous monitoring of relevant parameters (one example is gait pattern study, which has shown to see early parkinson onset).

Wearable devices are revolutionizing biomedicine, because they can provide:

- Continuous and longitudinal patient monitoring
- Automated health event prediction, prevention and intervention
- In-home chronic disease management

As engineers, we can use our knowledge in every field. The medical field is sometimes not so welcoming of technology.

Wearable systems can be bracelets, sensors embedded in clothes, smartphones; the location of the sensor depends on what we need to do with them. It is better to put it next to the center of mass.

Unsupervised settings make for more realistic data, but less knowledge about the collected data. How can we approach this? Two different approaches. Either ambient sensors, which have high installation costs and are limited to a specific environment, or wearable sensors which are faster, easier, their set up is at a lower cost, and they are ubiquitous.

Requirements for wearable sensors: it's easy for the user, because they just have to wear them, charge them etc so the first element is automation; the second one is reliability, whilst the third one should be usability as in comfortable and that should not change a person's lifestyle.

Sensors that can be embedded are inertial measurement units like accelerometers, gyroscopes, magnetometers, or barometers, or heart rate, or GPS, or microphone and camera.

One big problem is power consumption. Gyroscopes and magnetometers, compared to other sensors, consume more power than accelerometers and barometers. We must be cautious in using the right sensors only.

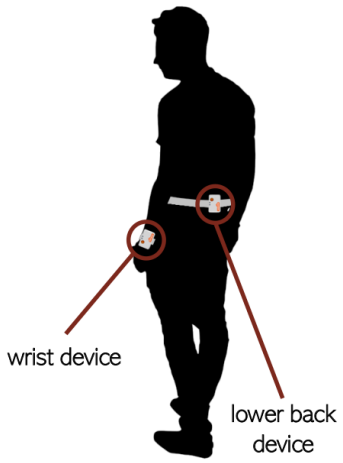
Accelerometers measure acceleration by measuring the current on the axis. We have a capacitor per axis and the result is an acceleration in a plot. The acceleration is measured in g. We have three axes, and we must choose carefully which axis goes upwards. When there is no clear solution, we can use the magnitude of the acceleration, which works in spite of the orientation of the device on the user's body.

If you monitor the gait cycle, sensors can let you measure a huge number of them, and machine learning can let us trace similarities in between them.

Now, a real case study: by studying gait patterns, we have studied how frailty can be observed and influences the way older people walk.
Frailty is an age-related condition characterized by an increased vulnerability to external stressors and an excess risk of poor clinical outcomes. The typical clinical assessment tool is the phenotype model, a set of five tests which puts the patient in one of three situations: robust, pre-frail, frail.
This condition should be prevented, because it is not reversible. Recent studies showed that there are several correlated gait pattern parameters highly correlated with frail and pre frail conditions.

IDEA: can we use machine learning on the gait pattern data to assess whether there is a correlation or not?

In the first phase the need was to assess if sensors can assess frailty, which are the most important parameters and which position is the best for the device.
So, we collect signals, preprocess them, then we detect the gaits, we find the steps and split the signal trace in different segments to avoid noise. We want an instance long enough to have all the information, short enough to cut out the noise. By doing this, we detect the gait segments. After this, we use machine learning and artificial intelligence to extract features by dealing with numbers. In the end, we see whether it is possible to assess frailty.
Here is the dataset information:



- 34 older adults aged 70+
- Subjects are screened according to Fried's phenotype
  → 23 NR (8 frail + 15 pre-frail)
  → 11 R (robust)
- Signal collected during 60m of regular walk
- Wearable devices:
  - Shimmer3 worn at the lower back
  - Shimmer3 worn at the wrist

In the dataset, everytime we have a peak we have a step. Preprocessing was done by applying a low pass filter to filter out non-body information (almost all human body information is under 20hz).
So, after preprocessing all peaks get detected, and segments are taken every 8 steps. So, the final dataset is made by segments of 8 steps each to have more data in smaller pieces and to be able to run a machine learning algorithm through it.
Of course, when programming we deal with numbers and timestamps, not with a graph.
We can also go from time domain to a time frequency domain by using the Continuous Wavelet Transform. We can clearly see the difference between the two kinds of subject by using this method.
We would like to develop a classifier to distinguish between the segments of different conditions.
It is a two steps classification: 1. Each of the subject's gait instances is classified as R or NR
2. The subject is classified according to a majority voting scheme.

Performance metrics are Sensitivity (proportion of actual frail people detected), Specificity (measures the ability of discarding robust subjects) and Accuracy. In medical fields, Sensitivity is the most important thing.

This experiment was done in a hospital setting, so it was supervised. Still, there was no interference by medical professionals.

What if we move out of the hospital?

In the first part of this study, we have investigated the use of wearable sensors to analyze gait and identify pre-frail and frail subjects in a supervised environment
Next step is to evaluate our method in a longitudinal study, in order to assess the feasibility of the proposed approach in uncontrolled environments.
New experimental setup:

- Only one device: wrist-worn Shimmer3 (most comfortable and gave better results prior)
- 24-hours data collection on each subject
- In-home environment

What happens now? We have 35 subjects who wore the device for 24 hours and then gave the bracelet back.
The data was downloaded from the sensor, the trace was extracted in csv format and loaded into a Pandas dataframe.
Preprocessing included magnitudes computation, and signal cleansing with the same low pass filter as before.

Now, after the 24 hours trace collection and signal processing, we need to segment the dataset and detect the gait pattern.

Segmenting is needed because we have much more data. So, for each subject the whole signal was cut into 5 second segments, and all the segments with less than 4 steps have been excluded. 5 seconds is the average time duration of 8 steps. This cleans the dataset a lot, and empties it of all noise.

Feature extraction: this is the same thing that was done in the first part of the study, so CWT applied for the 5 second segments, and then a stage of feature extraction to represent the data through a vector of 128 features.

Still, we are detecting a lot of noisy gait, because the experiment was unsupervised. So, dimensionality reduction was applied for each subject, where the N most active segments were selected, i.e., those segments with the highest signal energy value (i.e., RMS) The best N was 300.

Features were selected using the training set and the numerosity of the dataset was diminished, because of the limitations of some models.

How was the classification done? The validation methods used were Leave one subject out cross validation; feature extraction applied on the entire dataset and feature selection applied only on the training set; ML models tested using the validation set.

Classification was still done in two stage process where 1. Each of the subject's gait instances is classified as R or NR and 2. The subject is classified according to a majority voting scheme.

The results, obtained with a Gaussian Naïve-Bayes classifier:

ACCURACY = 0.77, SENSITIVITY = 0.84, SPECIFICITY = 0.69

Accuracy is not so good, but sensitivity is very good here and for healthcare professionals is the most important indicator.

Conclusions: it is important and easy to create tools to help reshape healthcare. Wearable devices are very important, just like medical care in a home environment. Sharing the data is important both for the patient and for the medical professionals.
Computer science and engineering knowledge can make the models work in an easier way.

LECTURE 11: 05/04/2022 (Vecchio)

From the register: two case studies about sensing the human body and sensor fusion in the authentication domain

the recording on the channel was not accessible and i could not follow this lecture

LECTURE 12: 06/04/2022 (Avvenuti)

From the register: Pastry: overlay network, prefix routing, routing table, locality, node arrival/departure, API. Scribe: rendez-vous point, multicast tree, publish-subscribe implementation, topic creation, subscription, message delivery, tree repairing, API.

this part is done thanks to the help of the notes of Tommaso Amarante, which helped me very much because i couldn't attend these lectures nor did i find the recordings.

Let's keep talking, briefly, about DHT operations. A typical use of the DHT for storage and retrieval might proceed as follows:

1. to index a file with given filename and data in the DHT, the SHA-1 hash of filename is generated, producing a 160-bit key k, and a message insert(k,data) is sent to a node participating in the DHT;
2. The message is forwarded from node to node through the overlay network until it reaches the node responsible for key k as specified by the keyspace partitioning. That node then stores the key and the data;
3. any other client can then retrieve the contents of the file by again hashing filename to produce k and asking any DHT node to find the data associated with k with a message lookup(k);
4. the message will again be routed through the overlay to the node responsible for k, which will reply with the stored data.

So, first we produce the hash of the file name, which becomes the key, and then we send the key-value pair to a node, and this message is forwarded to the node which is responsible for that key.

Any client can retrieve the data by hashing the filename, getting the key and searching for it with the lookup(k).

The properties of DHT are:

- **Autonomy and decentralization**: the nodes collectively form the system without any central coordination
- **Scalability**: the system should function efficiently even with thousands or millions of nodes
- **Fault tolerance**: the system should be reliable (in some sense) even with nodes continuously joining, leaving, and failing
- **Load Balancing**: the hash function should evenly distribute keys to nodes
- **Anonymity**: can be achieved by special routing overlay networks that hide the physical location of each node from other participants

DHT scales to huge numbers of nodes, and can handle both failures and any kind of variation in the number of nodes, thus DHT can be used to build more complex services (peer to peer file sharing, instant messaging, content distribution systems…) in IoT, DHT-based peer to peer architectures can be used for discovery service and to manage the flow of data from sensors to the applications.

With the DHT the P-S approach is simpler since we do not have to flood the entire network. The complexity for a lookup message is O(log(N)), in this way a subscriber reaches the repository node in a more efficient way.

Pastry: it is a peer-to-peer overlay network for DHT. In this method the key-value pairs are stored in a peer-to-peer network of internet hosts in a redundant way, and redundancy guarantees fault tolerance.
With this method we need to design a routing schema that avoids the use of flooding.
At the bootstrap, pastry starts with the translation of the IP addresses of the participating nodes, which is done using a secure hash function that produces a fixed length integer that is called NodeID.
At bootstrap the system has also to build the routing table for each participant in order to implement the routing schema on the overlay network.
Because of its redundant and decentralized nature, pastry has no single point of failure and any single node can leave the network without warning with a little or no data loss.

Design: The most important thing is the rule that allows us to partition the key space, more precisely keys are stored in the node with the NodeID numerically closest to the key among all live pastry nodes. In general the topology of the overlay network is circular.
NodeIDs and Keys are 128-bit unsigned integers represented in base 2^b, the integer represents the position in the circular space.
NodeIDs have to be chosen randomly and uniformly, in this way peers which are adjacent in NodeID are geographically diverse (maybe distant or not: the overlay network is a logical structure so, nodes that are adjacent there are not necessarily physically close to each other).

Routing: given a message and a key, Pastry routes the message to the node with the NodeID that is numerically the closest to the key.

We must be able to jump from a node to another in order to go as far as possible (if we had to travel the whole network using adjacent nodes, if N was the number of nodes the complexity would be O(N)) and by using pastry we are able to route to any node in less than $log_{2^b}N$ steps on average.

Prefix routing: in Pastry, the routing is based on the “Prefix Routing Schema”.
As we said before, the NodeIDs and the keys are sequences of digits in base 2^b.
The routing table is organized into $log_{2^b}N$ rows, and 2^b columns. So, if b = 1, we represent the NodeIDs in a binary way and we will have two columns.
The entries in row n refers to a node whose NodeID matches the present node’s NodeID (the NodeID of the host of the table) in the first n digits, but whose (n+1)th digit has one the $2^b - 1$ possible values other than the (n+1)th digit in the present’s NodeID (and on the subsequent digits whatever). (Basically, I check my node’s NodeID and for each row n I insert the NodeIDs which have the first n digits equal to my NodeID and the n+1 digit is different, and every column is one of those values). Since each entry refers to one of potentially many nodes whose NodeID matches the actual prefix, we have to choose among them the one that is physically closest to the present node using a proximity metric like the round trip time.

For example, a routing table whose NodeID is 65a1x and b = 4, x is an arbitrary suffix… here it is:

Routing Table of a Pastry node with:

nodeID = 65a1*x*

b = 4

- digits are in base 16
- *x* represents an arbitrary suffix
- the IP address associated with each entry is not shown

0 x	1 x	2 x	3 x	4 x	5 x		7 x	8 x	9 x	a x	b x	c x	d x	e x	f x
60 1x	61 1x	62 1x	63 1x	64 1x		66 1x	67 1x	68 1x	69 1x	6a 1x	6b 1x	6c 1x	6d 1x	6e 1x	6f 1x
650 1x	651 1x	652 1x	653 1x	654 1x	655 1x	656 1x	657 1x	658 1x	659 1x		65b 1x	65c 1x	65d 1x	65e 1x	65f 1x
65a0 x		65a2 x	65a3 x	65a4 x	65a5 x	65a6 x	65a7 x	65a8 x	65a9 x	65aa x	65ab x	65ac x	65ad x	65ae x	65af x

The separations indicate a row, and the columns inside that separation are the possible values of the row.

In the first row (which is the row with n = 0) there are all the prefixes that do not share digits with the NodeID of the present node. As we increase the row, of course, we will have that an incremental number of digits match the NodeID of the present node.

In the second row (which is row 1 since we start from 0) in the picture we can have 6 as the first digit; then, the second digit can be anything but not 5; then, all the other digits can be whatever.
If in my network I have a lot of nodes that match the prefix, how can I choose the one to put in that entry? I can use a proximity metric, so I should choose the nearest node in the physical world.

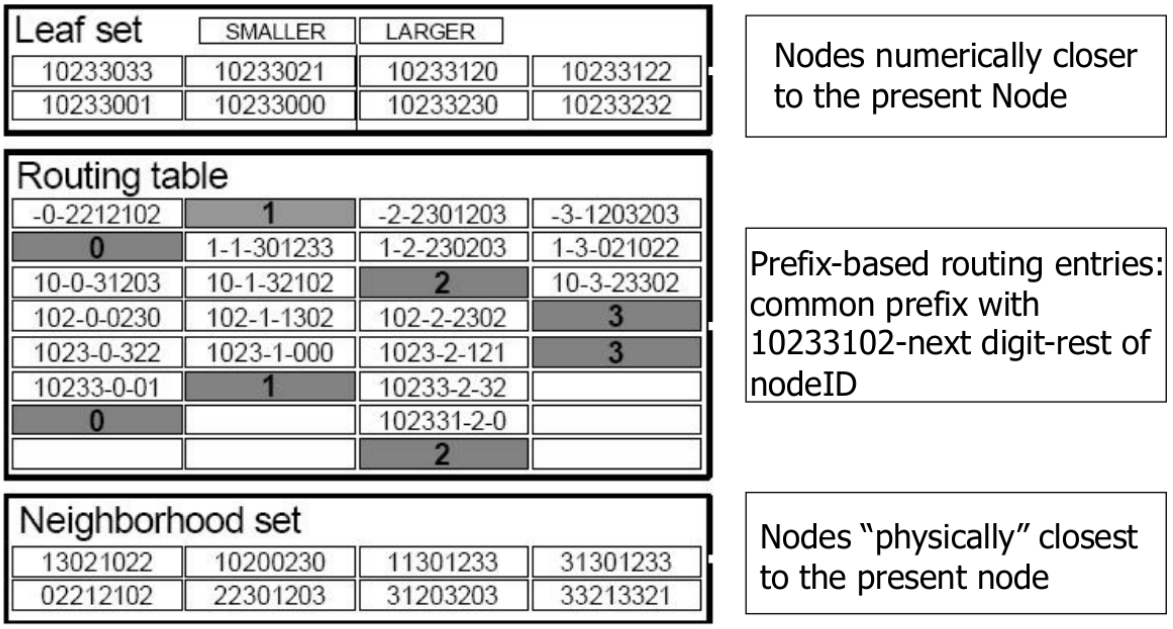
Another property is that in the first row we store nodes that are logically far (not physically) from the present node, so when we choose one of these NodeID as destinations we will make the longest jump in the overlay network.
On the other hand, when we choose a NodeID from the last row we will make the shortest jump: this means that we are closer to the node that stores the key that we are looking for.

Since b is equal to 4 the NodeIDs are expressed as hexadecimal numbers. In this table the IP addresses are not shown, we have to store them since we have to route the messages.

So, in addition to the routing table, each Pastry node stores a leaf set. In this set there are the NodeIDs of the nodes that are numerically closer to the current node. In particular, in the leaf set we store the

- l/2 nodes with numerically closest larger nodeIDs
- l/2 nodes with numerically closest smaller nodeIDs

Each node of the network maintains the ip addresses of the nodes having nodeIds inside its leaf-set.



nodeId=10233102, b = 2, l = 8

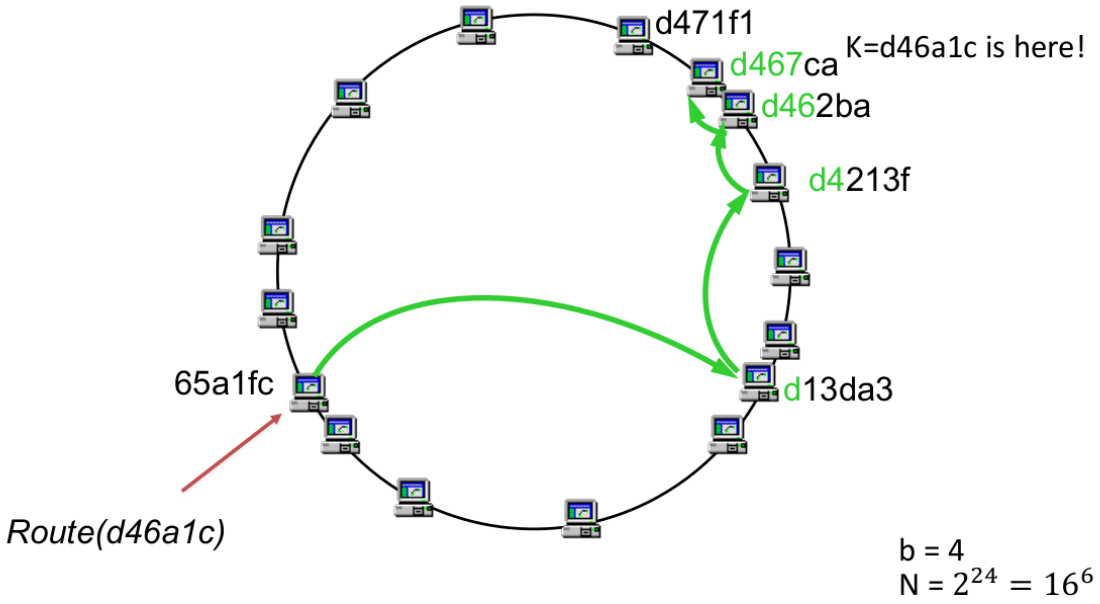
This picture depicts a complete table.
We can see that in this example l is the length of the NodeIDs (this results in a key space of 2^16, since every digit requires 2 bits to be expressed in base 4) so we are storing, in the leaf store, 4 nodes with lower NodeIDs and 4 with larger ones. It is important to notice that because of the parameter every node in the leaf set shares 5 digits with the present node, from 0 to 4.

In the neighborhood set are stored the NodeIDs of nodes that are physically closer to the present node. This set is useful for dealing with redundancy and with failing nodes, for example when the node is going to fail, then the node itself tries to upload its keys on the near nodes.

Routing steps: this is how Pastry handles the full routing procedure.
Let's suppose that a node receives a lookup message. This message has the key of the target node as a parameter.

1. Seek the routing table (we need this step because the current node must forward the message to a node whose NodeID shares with the key a prefix that is at least one digit longer than the prefix that the key shares with the current NodeID, and we do this because each key must be stored in a node whose NodeID is numerically closest to the key - the further they are, the longer the jump)
2. If no such node is found in the routing table, check the leaf set (if we do not find a node in the routing table, that means that we are close to the target node! So, by looking at the leaf set, we are looking at nodes whose NodeID shares with the key a prefix which is as long as the one on the current node, but it is numerically closer - with this step we will perform short jumps)
3. If no such node is found in the leaf set, the key is stored in the current node. (unless the $l/2$ adjacent nodes in the leaf set have failed concurrently).

This method guarantees a complexity of O(log(N)) in the number of jumps.



Locality: as we already said, when we have more than one choice for a NodeID in a specific entry of the routing table, Pastry always selects the physically closer one. In order to do this, we use a proximity metric like round trip time, that reflects the distance between nodes using a scalar value. The locality is ensured by choosing a function that allows each Pastry node to determine the distance between itself and a node with a given IP address.

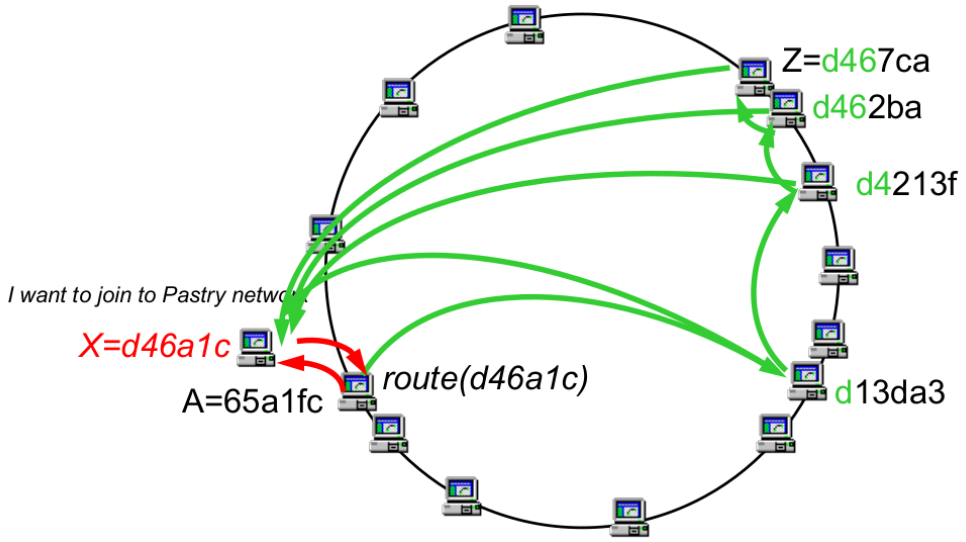
Short Routes Property: Given the locality assumption, we can assume that in the Pastry route mechanism each node in the routing table is chosen to refer to the nearest node, according to the proximity metric, with the appropriate NodeID prefix.
As a result, in each step a message is routed to the nearest node with a longer prefix match. Simulations show that the average distance traveled by a message is between 1.59 and 2.2 times the distance between the source and the destination in the underlying Internet.

Route Convergence property: this property compares the distance that two messages that refer to the same key travel before their routes converge. We also assume that the two messages, called lookup messages, have different injection points in the network. Simulations show that, given a network topology, the average distance traveled by each of the two messages is approximately equal to the distance between their respective source nodes.

Node Arrival: if a new node arrives, since the underlying network is the Internet, the IP address is known. Because of that, we can use the hash function and retrieve the NodeID of the new node. We must then put the node in the portion of the ring where there are the nodes with the NodeID numerically closer to the new one.
Since the problem is equal to the lookup for a key in the network, we can use the same routing schema. The only difference is that the routing table of the new node must be initialized and we have to inform the adjacent nodes that the node is inserted, in particular to insert a new node whose NodeID is X, we follow the next steps:

1. We contact a known nearby Pastry node A, according to the proximity metric.
2. We ask it to route a “join” message using X as a key;
3. The message is routed to the existing node Z with the NodeID numerically closest to X
4. X obtains the leaf set from Z
5. The i-th row of the routing table of X is taken from the i-th node encountered along the route from A to Z (basically, i copy the i-th row of the routing table of the i.-th encountered node in the routing table of the new node)
6. X notifies the nodes to update their states

Node Arrival Example



What if a node departs or fails? Neighboring nodes in the NodeID space (more like, in the leaf-set, since they have each other's ip addresses) exchange periodical keep-alive messages. We assume that a node fails if it is unresponsive for a set period of time T.
If this happens, all the members of its leaf set are notified, and they update their leaf sets.
Since the leaf sets of nodes with adjacent NodeIDs overlap, this update is easy.
If the node recovers, it will contact the nodes in its last known leaf set and obtain their current one. After this, the node notifies the members of its new leaf set of its presence.
Routing table entries that refer to failed nodes are repaired lazily (it takes some more time).

What about the Pastry API? we need to ensure the functioning of several functions:

- nodeId = pastryInit(Credentials) causes the local node to join an existing pastry network, or start a new one, and returns the local node's NodeID. The credentials are provided by the application and contain info needed to authenticate the local node and to securely join the Pastry network
- route(msg,key) causes pastry to route the given message to the node with the nodeID numerically closest to the key
- send(msg, IPaddr) causes pastry to send the given message to the corresponding node, if it is alive

Applications layered on top of Pastry must export the following operations:

- deliver(msg, key) it is called by Pastry when a message is received and the local node's NodeID is numerically closest to the key among all the live nodes - or when a message is received via the send, using the IP address of the local node
- forward(msg, key, nextId) is called by pastry just before a message is forwarded to the node with nodeId = nextId. the application may change the content of the message or the value of nextId. Setting the nextId to NULL will terminate the message at the local node.

- newLeafs(leafSet) is called by pastry whenever there is a change in the leaf set. This provides the application with an opportunity to adjust application-specific invariants based on the leaf set.

Now, let's talk about Scribe. It is a publish-subscribe system based on Pastry. Just to recall, publishers stream content and subscribers can choose to receive it.
In Scribe, any node may create a group, and other nodes can then join it or multicast messages to all members. The dissemination of the messages in the group is best-effort (that is to say, it will always be attempted, but without the guarantee of success) and there is no particular delivery order.

Scribe a peer-to-peer network of Pastry nodes to manage group creation, group joining and to build a per-group multicast tree used to disseminate the messages multicast in the group.
It is fully decentralized: all decisions are based on local information, and each node has identical capabilities
Scribe can support simultaneously a large number of groups with a wide range of group sizes, and a high rate of membership turnover.

API:

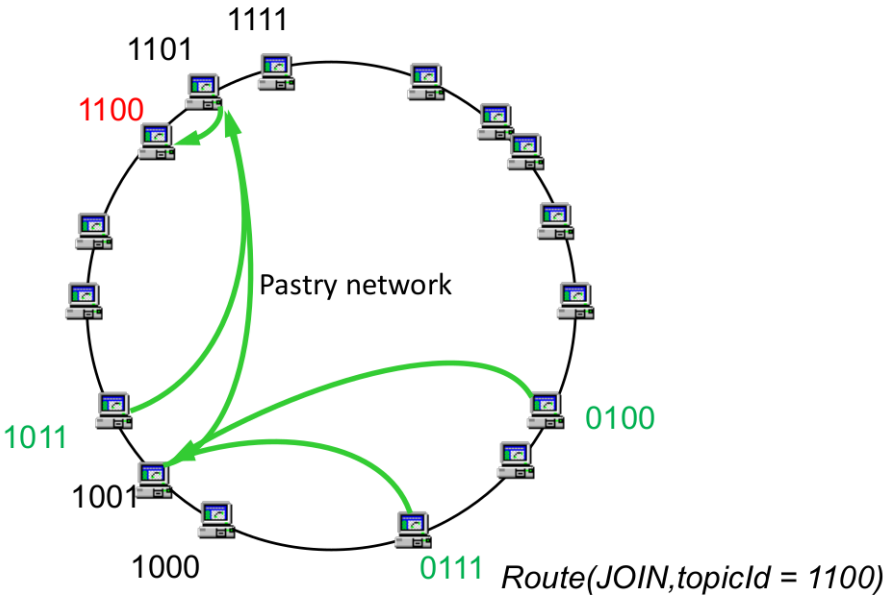
- create(credentials, groupId) creates a group with groupId. Throughout, the credentials are used for access control
- join(credentials, groupId, messageHandler) causes the local node to join the group with groupId. All subsequently received multicast messages for that group are passed to the specified message handler. it can be used by the subscribers
- leave(credentials, groupId) causes the local node to leave the group with groupId
- multicast(credentials, groupId, message) causes the message to be multicast within the group with groupId. it can be used by the publisher.

Implementation of the publish-subscribe: each topic (group) has an unique topicID, that can be generated from any sentence or string defining the topic, using the pastry hash function. So, the topic id will belong to the pastry keyspace.
The rendezvous point is a Scribe node with a NodeID numerically closest to the topicID, as well as the root of the multicast tree that will be used to deliver multicast messages to the group. It is important to mention that the Scribe software on each node provides the forward and deliver methods that are invoked by Pastry whenever a Scribe message arrives.

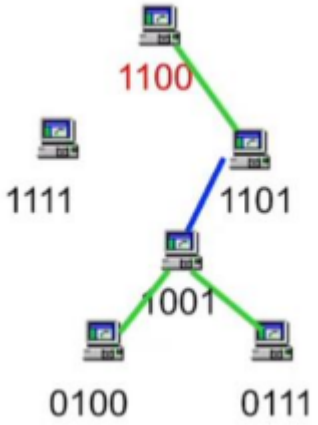
The topicID is the hash of the topic's textual name concatenated with the name of the creator. The hash is computed using a collision resistant hash function, which ensures a uniform distribution of topicIDs. Since Pastry nodeIDs are also uniformly distributed, this ensures an even distribution of groups across Pastry nodes.
Now we will see how to create a topic. The route message is sent to a node by the publisher. One of the inputs for the route function is the create function, which requires credentials and the topicId (of the scribe's api). Then, pastry works with its routing to find the rendez-vous point, so the Pastry node that will store the topicID. The rendezvous node will receive the route message and call the deliver method implemented by scribe, to let scribe know that it is storing a given topic.

The Route and Deliver methods are part of the Pastry API, whereas the Create method is part of the Scribe API. Deliver is called to receive the message when the node-id of the current node is the closest one among all live nodes.

Scribe creates a multicast tree, rooted at the rendezvous point, to disseminate the multicast messages in the group. It is created by using a scheme similar to reverse path forwarding. The tree is formed by joining the Pastry routes from each group member to the rendezvous point.



So, we have the pastry layer and the scribe layer side by side. (look at picture)
If we have a topic and it is stored at the rendezvous point 1100 (in red) and we have new subscribers, and the first asks to join the group. The method is route and the message is JOIN. So, it routes the join message based on the routing table until the rendezvous point is found. We simply have to look up where the key of the topic is. Once the message reaches the rendezvous point, we have built a multicast tree rooted at 1100.



The leaf nodes of the multicast tree are the actual members of the group.

When we need to disseminate a message, the publisher will send a message to the rendezvous point, which knows the first hop in the multicast tree, through the Pastry network using the TopicID as destination. Once the root has received the message, it knows the destinations and the first hop in the multicast tree. It simply forwards a copy of the message to the first hop and then the messages propagate according to the tree's structure.

Fault tolerance: the multicast tree can be repaired. It may happen that a link between nodes breaks,

fortunately the multicast tree is fault tolerant. For example, the broken link is the one in blue in the image.

To solve the problem, the two leaf nodes 0100 and 0111 can send the Join request again, in order to find an alternative way to the root node. The tree, after this, might change structure.

LECTURE 13: 12/04/2022 (Avvenuti)

From the register: Humans as Sensors: participatory and opportunistic crowdsensing, the HaS paradigm, the Reliable Sensing Problem. A Binary Model of Human Sensing: Unknown Reliability, Binary Observations, Uncertain Provenance.

Today we start talking about another part of the course, which is social sensing, which means using humans as sensors. For example, in social networks it is the people themselves who give out information about their environment, and more. The idea is to use social platforms as sensor networks.
Of course, we face some problems - are we sure the information is true, for example?
We are going to talk about cases when people post something about a real life event that happened.

Mobile crowdsensing: large group of individuals having mobile devices capable of sensing and computing collectively share data and extract information to measure, map, analyze, estimate or infer any processes of common interest.
The idea is to extract user information from these collective sources. Of course, we do not exclusively consider human sensors. Humans carry physical sensors in the smartphones themselves, and we have opportunity to have information from the environment - at the beginning of exploring this idea, the fact that humans produce data was not considered.

Based on the type of involvement from the users, mobile crowdsensing can be classified into two types:

- Participatory, where the users voluntarily participate in contributing information
- Opportunistic, where the data is sensed, collected and shared automatically without user intervention and in some cases, even without the user's explicit knowledge

There are also hybrid systems, that start from opportunistic and move to participatory.

We move on to talk about the Humans as Sensors paradigm: as we told before, we want to consider information that comes from humans, not from the physical sensors. The huge availability of social network content suggests that the largest “sensor network” yet might be human. Humans as Sensors can be considered an extension of participatory sensing, because we need to use social networks as sensor networks where humans represent sensors.
The occasional observations they make about the physical world represent (data) claims.
We will talk about the activity of humans on social media during emergencies. There are many examples of twitter usage immediately after an emergency before any official news reported anything.

With traditional sensors, it takes a longer time to discover, for example, an earthquake. Also, social networks can give immediate information about specific damages caused by a huge event.

HaS: the reliable sensing problem. When we use humans as sources, we know that they aren't always reliable.
Let consider only participatory sensing of the physical environment that is external to the human sensor (e.g., «Shooting erupts on Liberty Square!»)
Subjective reporting such as «It is an inspiring day!» and personal emotions «I am depressed» are not considered, in order to simplify the problem.
In this case, the physical environment has a unique state, leading to a unique ground truth, according to which human observations are either true or false (e.g., whether there was shooting on Liberty Square or not) We will call this problem the "Uncertain provenience of observations".

Observations can be considered either true or false. The sensing problem is to determine and understand which claims are correct, given that the reliability is not known a priori and the provenance of reported observations is uncertain (i.e., individuals may report observations made by others as their own).

Let's start to define a model that we can use to treat the problem - this one is called the Binary Model of Human Sensing.
When we use observations for humans we know that the sources are not reliable. So, model human participants as sources of unknown reliability generating binary measurements of uncertain provenance.
What we can do is try to formulate a rigorous estimation-theoretic problem to optimize filtering of correct observations in a maximum likelihood sense - that is to say, we try to understand to which extent the observation we use is reliable.
Let demonstrate that this model enables the reconstruction of ground truth from noisy human observations in practice.
In other words, we make an observation that is not necessarily true, we can use a distribution of probability in order to assume the probability of the observation being true (failure model).

What about unknown reliability? We have already discussed this point, we can say that with a physical sensor we know exactly its characteristics: for example, range, calibration, failure modes of the sensor. With a human we do not have all this information nor can we act to modify properties.
Another point is that, for example when we use an accelerometer, we know that it always measures the acceleration; we do not know if the human sensors are willing to provide an exact measure, and always the same type of measure. Also, humans are much broader, though less accurate, in what they can observe.
Finally, different individuals have different levels of reliability.
So, the reliability is the probability of producing true claims.
A true claim is one that is capable of reproducing the physical world's state.

What about binary observations, then? The physical world is a collection of facts that people want to report.

In our example facts are restricted to some kind of emergency (e.g. “Main street is flooded” - it’s true, different humans might report it in a different way, but it either is true or it is not, “The BP gas station on University Ave is out of gas”).
Human sensors report some of the facts they observe.
Since the observation can either be true or false, we can consider claims as measurements of different binary variables, as we know that we can associate a probability distribution to a binary variable when we are not sure about the reliability of the source.
So, reliable sensing means inferring which among the reported human observations match ground truth in the physical world.

What about uncertain provenance? When, for example, we analyze a tweet that states “Main street is flooded”, even if we authenticate the Original Poster as the actual source of the tweet, we still do not know if they truly observed that first-hand, or if they heard it from somebody else.
We can state that because it not unusual for a person to report observations they received from others as if they were their own. This, of course, does not occur with physical sensors. From a sensing perspective, these kinds of errors in measurement across sensors may be non-independent. Because of that, when you post a non-fist-hand false observation you mystify the physical world in a non-independent manner.
One erroneous observation might ìt be propagated by other sources without being verified.
If we want to know whether a set of claims are true or not, we can for example use the vote approach. This means that we can consider a claim “true” if a large number of users have made that claim.
We can easily understand how this approach is not correct, since many users can repost a false claim. This would result in a high number of votes even if the claim if false.

Let’s move to the solution. We must unreliability of humans, the uncertain provenance of observations, the fact that we do not know the ground truth, and we must understand the parameters that characterize the phenomenon.
We can use an architecture formed by different software modules that can be implemented in order to solve this problem:

- Collect data from the “sensor network” - so any social media like twitter or facebook.
- Structure the data for analysis: the data we can extract from social media is unstructured, texts are mostly not very structured grammatically speaking, and in general content does not follow a standard.
- Understand how sources are related: we have to understand how sources of information are related together, we have to understand for example if there is a network that describes relationships between humans when they talk about the same real phenomenon, this is because we want to know if they are related or not. This point deals with the problem of uncertain provenance.
- Use this unstructured information and make it structured - for example, translate it into vectors in a given space, so that at the end of this phase we have clusters of observations that talk about the same phenomenon.
- Finally we can use the likelihood obtained from the probability distribution in order to classify an observation as true or false.

First step is the data collection - we can use any platform, twitter is very easy if we use APIs and some keywords. This is because Tweets are collected through a long-standing query via the Streaming Twitter API to match given keywords and, optionally, an indicated geographic region on a map. These can either be and-ed or or-ed.

At the beginning of this phase we have a collection of non-structured text that we extracted from twitter.
First of all we can use some NLP tools (also process the semantic) to process each tweet. We can use the tokenize function, that can extract a word or a group of words from a text.
By doing that we discard useless words and we obtain a single vector for each tweet in our collection.
After this procedure, we can apply a logical distance function to each pair of structured tweet (vectors containing tokens): in this way we can obtain a similarity measure between two tweets, the more dissimilar the larger the distance. We can use the cosine similarity function to measure the similarity.

$$\text{similarity} = \cos\theta = \frac{A \cdot B}{\|A\|\|B\|}$$

dot product

vector magnitude

So, we collect tweets. Users are not reliable. Observations are either true or false, and many talk about the same thing. For each observation, using NLP tools, we create a vector. Vectors may report the occurrences of the least of tokens that represent the cardinality of the space we are considering. Then, we use the cosine similarity function to return a measure of similarity based on the number of matching tokens in the two inputs.
Now we have space, tokens, many vectors, for each pair we have the similarity.

We can now build a graph where vertices are the individual observations (that is to say, the tweets) and the links represent the similarity among them - the value of the similarity is the weight of the link. We then cluster the graph, causing similar observations to be clustered together, and we call each cluster a claim - defined as a piece of information that several sources reported.
Usually, you do not need to cluster data coming from physical sensors.

We call this graph a Source-claim graph. Sources are humans, claims are clusters. Each source is connected to all the claims they made (that is, to all the clusters they contributed to), each claim is connected to all sources who espoused it (so, sources of tweets in the corresponding cluster).
Each claim is then considered true if it is consistent with ground truth in the physical world, otherwise it is false.

Now, we must talk about source relationships. By looking at the SC graph, we cannot determine whether sources are independent. They may or may not have reported observations they heard from others.
We assume the existence of a latent social information dissemination graph, SD, that estimates how information might propagate from one person to another.

So, it basically depends on how users interact on social media - users in twitter for example can either produce their own tweets or retweet other tweets. Retweeting means reporting information that are not your own and amplifying its engagement.
A more general method is called Epidemic Cascade network: each distinct observation is modeled as a cascade, and the time of contagion of a source describes when the source mentioned this observation. So, we can see who said what first, and how fast and where it propagated.

We can build social dissemination graphs using other informations and other kinds of networks, like:

- Follower-followee network: a direct link from A to B exists in the social graph if A is B is a follower of A
- Re-tweeting network: a direct link from A to B exists in the social graph if B retweets a tweet from A
- Combined network: a direct link from A to B exists in the social graph either if B retweets a tweet from A or if B follows A.

We use these four kinds of networks to map the dissemination of information.

LECTURE 14: 13/04/2022 (Vecchio)

From the register: Some Android libraries: voice-based interaction, ML kit, an example about object detection. Firebase Cloud Messaging, solving the problem of Network Address Translation. Location and Maps. Network-based positioning. Position-related libraries in Android. Bluetooth Low Energy: physical layer, link layer. Broadcaster/observer, central peripheral. Attribute protocol.

Today we are going to see some useful libraries for implementing your app. This is just an overview of possibilities, we are just going to understand the main functionalities and see details for only a couple of them.

Voice based interaction can be helpful when a keyboard cannot be used.
There are two forms of voice based interaction:

- Speech to text and text to speech engines and functionalities
- Voice actions (voice commands to smartphones)

Voice actions are delivered as intents. There is a list of predefined actions, and there can be activities which need a voice confirmation from the user.

Google also gives to developers a Machine learning tool kit, that provides several functionalities, related to both video and picture and text based information.
For example, it can detect faces, barcodes, it can label images, detect objects and recognize text. It can identify languages, translate and provide a smart reply to texts. it can, of course, be customized via defining some models.

Barcode scanning is useful, because barcodes are a convenient way to pass information from the physical world to your app: barcodes can encode structured data such as contact information or Wi-Fi network credentials.
This library works with different formats of barcodes, and it can also automatically extract texts of documents, business cards, and driving licenses. It may also return the position of text blocks in an image.

Face detection and image labeling are two other powerful tools. The face detection API detects faces in an image or video, identifies key facial features and fetches the contours of detected faces.

Image labeling lists the recognized entities in a picture. All labels come with a score that indicates their confidence. The list of labels is quite long - it depends on if the application is on device (then labels are something like 400+ or on cloud (the labels are more than 10k).

The landmark recognition API is a specialization of the image labeling function, which recognizes landmarks and produces each landmark's geographic coordinates and the region of the image the landmark was found, going as far as automatically generating image metadata.

Object tracking lets us detect objects in images and videos, and track them. It provides a coarse grained categorization and it is a first stage of a multi-level detection system.

How do you use such an API? Here we have an example, with how Object Detection works.
The idea is that the Activity starts the Image Gallery to let the user select an image; then, the Image Gallery returns the selected image uri in an intent by means of the onActivityResult. The image is then processed by using the ML kit.

```
public void onActivityResult(int requestCode, int resultCode, Intent data) {
    super.onActivityResult(requestCode, resultCode, data);
    Uri uri = data.getData();
    FirebaseVisionImage image = null;
    try {
        image = FirebaseVisionImage.fromFilePath(this, uri);
    } catch (IOException e) {
        e.printStackTrace();
    }
    …
```

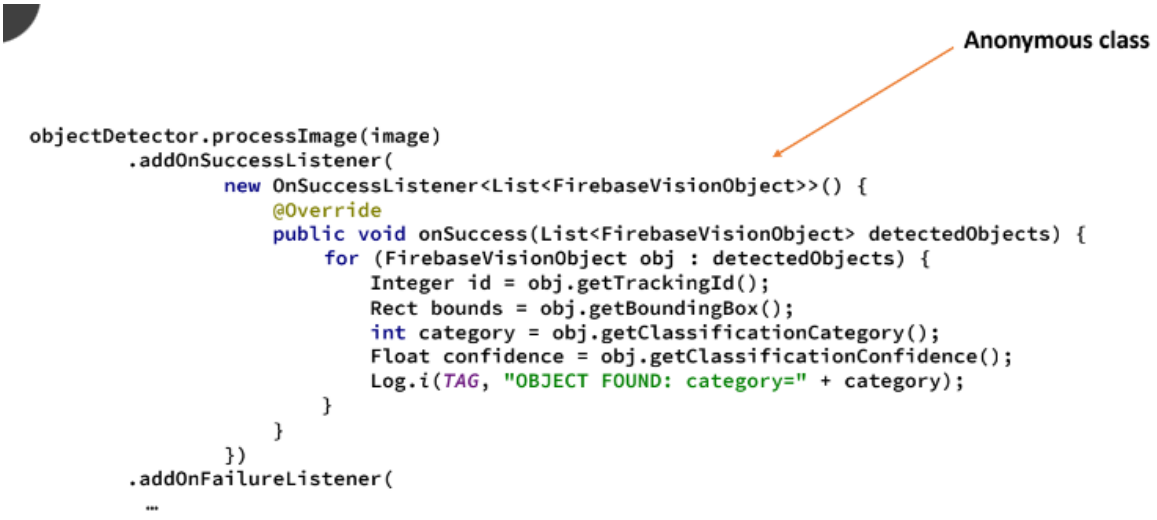
Path of selected image

Then we have to configure the object detector, and this is done by means of a helper builder class. This approach is followed in a series of APIs, but Android is not consistent in following approaches.
The idea is to create a builder object that helps us to create a FirebaseVisionObjectDetectorOptions object, which makes clear what we would like to have from the object detector.
Here we have a chain of methods, because each method returns the same object, so that the other methods can be invoked on the same object.
The first method specifies that we want to analyze an image and we are interested in detecting multiple objects, and we want the system to classify it. Then, thanks to this method, the options object is created.

This option object is then given to the getOnDeviceObjectDetector method that is applied on a FirebaseVision.getInstance()

```
FirebaseVisionObjectDetectorOptions options =
        new FirebaseVisionObjectDetectorOptions.Builder()
                .setDetectorMode(FirebaseVisionObjectDetectorOptions.SINGLE_IMAGE_MODE)
                .enableMultipleObjects()
                .enableClassification()
                .build();

FirebaseVisionObjectDetector objectDetector =
        FirebaseVision.getInstance().getOnDeviceObjectDetector(options);
```

Here we have the code that notifies the programmer of when an object is detected. The processImage method starts the classification and detection, and then we must apply two listeners (one for failure, one for success) to the object that is given back to the method.
The argument to this addOnSuccessListener is an instance of an anonymous class that provides an implementation of the onSuccessListener interface. That is why we have an implementation of the onSuccess method - in particular, its argument is a list of objects, which are the detected objects.

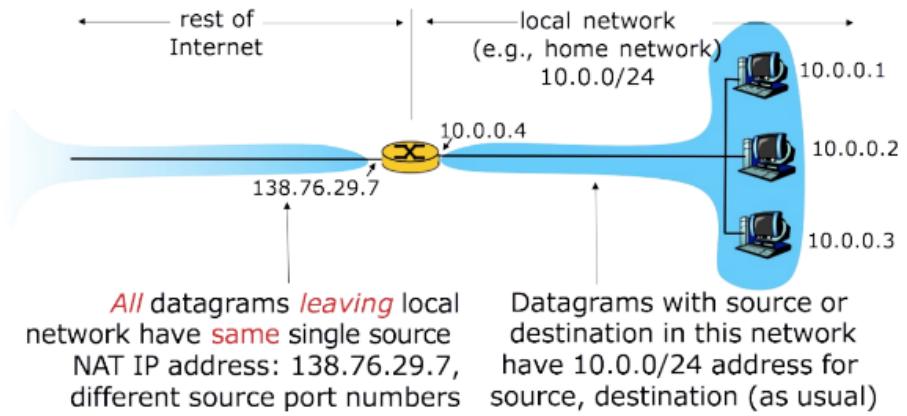


ML kit libraries are provided as part of Firebase: Firebase is a mobile application development platform, acquired by Google in 2014 and it provided two very successful products, Firebase Cloud Messaging and Firebase Realtime Database.
Now it also provides:

- Authentication
- Crash and performance analytics
- Cloud-based content hosting
- Remote configuration
- Advertisement

We are going to talk about Firebase Cloud Messaging. Which problem does it solve?
There are countless applications where we need to send information from a server to a client, an android application. The problem is that most smartphones are generally behind a Carrier-grade NAT.

- NAT?

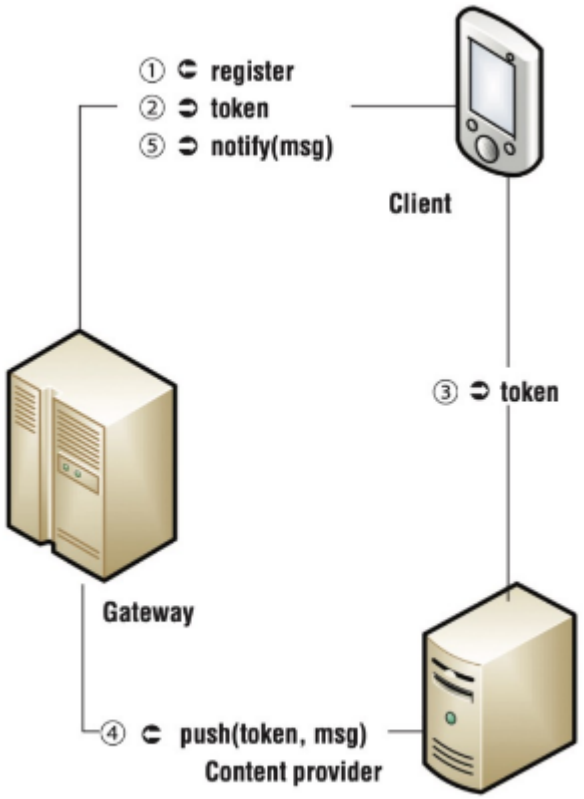


Carrier-grade NATs are used by MNO to cope with shortage of IPv4 addresses; the problem is that being behind a NAT, smartphones cannot be reached if they are not initiators of the communication.

Firebase Cloud Messaging in this situation is a cross-platform messaging solution that lets you send messages to smartphones, even when they are behind a net. It can notify a client app that new email or other data is available to sync.

It can send two types of messages:

- Data messages
    - you have to implement FirebaseMessagingService subclass
        - delivered to onMessageReceived() method
        - information is provided as a set of <key, value> pairs
- Notification messages
    - if app is in background a notification is raised, then app started when notification selected
        - if message contains also <key, value> pairs delivered as extra of starting intent
    - if app is in foreground, everything delivered to onMessageReceived()



Firebase provides a console in order to send messages. Some common ways you might receive such a message is for example when you hvìave applications on your phone and they send you a notification to alert that they haven't been opened in a while.

How can we connect our application with the source of information? First of all our app must register and get a token.
This token is provided to another component of our application - this is application dependent - and when we want to send a message to this device, you need to interact with the Firebase Cloud Messaging API, providing the message and the identifying token.
In order to counter the NAT, the device always has an open connection with Firebase and periodically sends a keepalive message.

Now, we'll talk about localization, APIs and general ideas. If we need to localize a smartphone we can use the GPS (accurate but only works outdoors) or wifi, access points and information coming from the cell tower signals. This technique is called location fingerprinting, and it is less accurate but useful indoors.

The accuracy of positioning, when obtained with a GPS, has an error of a couple of meters 95% of the time. So, it is reasonably accurate, but it requires line-of-sight between satellite and receiver, only works outdoors (signals don't penetrate buildings) and drains battery power.
Wifi location fingerprinting is more battery-friendly.
The idea is that At each (X,Y) location the set of WiFi APs observed and their signal strengths is unique, if we suppose that there are more access points in an indoor location.
We need a table containing pre-recorded tuples, to associate position with strengths; Google builds and stores this database (APs + Signal Strength) at each X,Y location and then all observed values (which will not be exactly the same as the ones already in the table) will be positioned next to the most similar tuples in the table.

How do you build the table? We can have devices with GPS and wifi simultaneously turned on, and send data to third party repositories (wigle.net) or Google. The GPS will collect the location, and the WIFI will collect the identifiers of visible AP and signal strength. This process is known as "war driving".
Another approach could be to collect data from the users that already are in the building.

Android has two location APIs, the new one , closed source and provided by Google Play Services and the old one being an Android framework location API.
We will take a look at the new one, which is more accurate because it takes information also from external sources, and this saves energy on the device. The other one is similar.

For privacy concerns, to use position you must ask permissions, in the manifest and at runtime.
• ACCESS_COARSE_LOCATION: precision=~city block
• ACCESS_FINE_LOCATION: precision=as much as possible
• ACCESS_BACKGROUND_LOCATION: when app in background

First of all, we need to obtain location: The new API fuses information coming from GPS and network for better precision and energy consumption. The app interacts with the FusedLocationProvider through a Client object.

```
private FusedLocationProviderClient myClient;

protected void onCreate(Bundle savedInstanceState) {
   ...
   myClient = LocationServices.getFusedLocationProviderClient(this);
}
```

Then we have to make clear the criteria that should be used when giving back the position in terms of accuracy and the rate to provide new positions.

```
LocationRequest myRequest = LocationRequest.create();
myRequest.setInterval(10000);
myRequest.setPriority(LocationRequest.PRIORITY_HIGH_ACCURACY);
```

This is a part of an Activity that starts and stops location updates.

```
protected void onResume() {
   super.onResume();
   myClient.requestLocationUpdates(myRequest, myCallback, Looper.getMainLooper());
}
protected void onPause() {
   super.onPause();
   myClient.removeLocationUpdates(myCallback);
}
```

Receiving updates requires an implementation of the LocationCallback interface, that may return several locations with timestamps.

```
myCallback = new LocationCallback() {
    @Override
    public void onLocationResult(LocationResult locationResult) {
       if (locationResult == null) {
          return;
       }
       for (Location location : locationResult.getLocations()) {
          // Use location information
       }
    }
 };
```

What if we want to obtain location updates in the background? The approach is quite different, because we need a foreground service, which is a service that is visible to the user because it shows an icon like a notification on the bar (we will use it for continuing user-initiated operations, e.g. suggesting directions in a navigator after user presses home button or turns off the screen).

Updates, when the app is in background, can be delivered at a reduced rate than what the app requested. Approach is based on PendingIntents, which are fired when new updates are delivered.
The first thing is to create an intent object and provide the name of the receiver class.
Then, the intent is customized by an action. Then, the intent is encapsulated in the PendingIntent.

```
Intent intent = new Intent(this, MyBroadcastReceiver.class);
intent.setAction(MyBroadcastReceiver.MY_ACTION);
PendingOntent pi = PendingIntent.getBroadcast(this, 0,
                                        intent, PendingIntent.FLAG_UPDATE_CURRENT);
myClient.requestLocationUpdates(myRequest, pi);
```

Now, whenever a new position is produced by the system, the application will be notified by means of an intent received by the MyBroadcastReceiver. This is done by the extractResult.

```
public class MyBroadcastReceiver extends BroadcastReceiver {

    static final String MY_ACTION =
            "mypackage.MY_ACTION";

    @Override
    public void onReceive(Context context, Intent intent) {
        if (intent != null) {
            final String action = intent.getAction();
            if (MY_ACTION.equals(action)) {
                LocationResult result = LocationResult.extractResult(intent);
                if (result != null) {
                    List<Location> locations = result.getLocations();
                    ...
                }
            }
        }
    }
}
```

Android also provides a geocoding and reverse geocoding API, which turns coordinates into geographic addresses and vice versa.

There is also a Places API, which provides contextual information about places (as in: physical spaces that have a name, like museums, bars, churches etc), which is not always free to use - depending on usage - and whose information on laces includes name of place, address, geographical location, place ID, phone number, place type, website URL, etc
This API can let us know the place where the device was last known to be located.

There are other location-related APIs:

- Directions API: calculates directions between locations (walking, driving, public transportation)
- Distance Matrix API: Calculate travel time and distance for multiple destinations
- Elevation API: Query locations on earth for elevation information, calculate elevation changes along routes
- Roads API: snaps set of GPS coordinates to road user was likely traveling on (best fit)

Almost all APIs require an API key.

if you want to include a map you need a key and you need to include a component to the SDK.

- There is a fragment class implementing a map
- An interface to be notified that map is ready
- Methods for adding markers, polylines, etc

```
public class BasicMapDemoActivity extends AppCompatActivity implements OnMapReadyCallback {

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

        SupportMapFragment mapFragment =
                (SupportMapFragment) getSupportFragmentManager().findFragmentById(R.id.map);
        mapFragment.getMapAsync(this);
    }

    /**
     * This is where we can add markers or lines, add listeners or move the camera.
     * In this case, we just add a marker near Africa.
     */
    @Override
    public void onMapReady(GoogleMap map) {
        map.addMarker(new MarkerOptions().position(new LatLng(0, 0)).title("Marker"));
    }
}
```

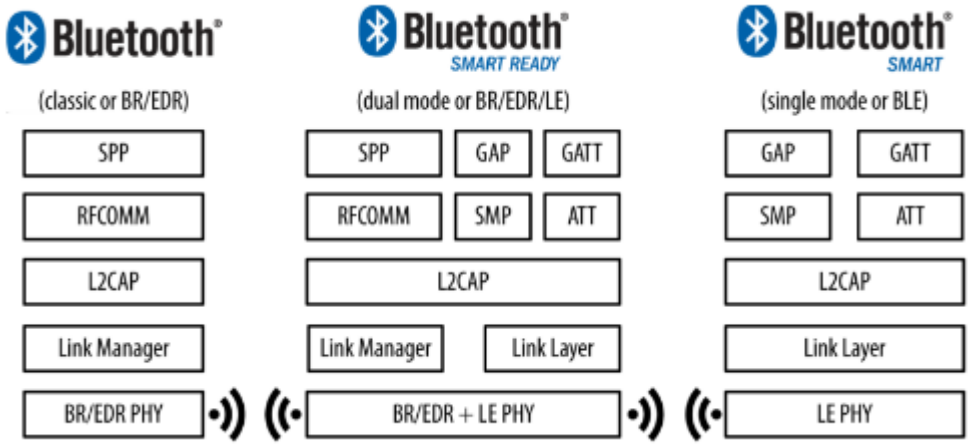
The fragment including the map can be included in a layout file. In the manifest you need to include the API_KEY, which can be obtained in Google developer console.

We can now move on to a different topic: Bluetooth Low Energy.
Bluetooth Low Energy (BLE), aka "Bluetooth Smart", is a lightweight subset of the Bluetooth 4.0 core specification. It is especially important in a mobile setting since it is an energy saving method.

Goal: design a radio standard with the lowest possible power consumption, specifically optimized for low cost, low bandwidth
• A coin cell battery may last for years
• $2 for a system-on-chip
• Typical throughput is in the order of 10-100 kbps

Smartphones and tablets greatly contributed to the massive adoption of BLE: users are familiar with using their smartphones, the UI is rich, and there is a low barrier to the adoption of devices that talk to smartphones.

BLE and classic Bluetooth are not compatible - we will focus on BLE since it is more relevant for our purposes.



In terms of architecture, there are several possibilities.
The first one is to include everything onto a single chip - controller part, hardware, application host, application logic. This is what you find in cheap devices.
Another approach is to have part of the implementation on software - this is the approach followed by smartphones, where there is a controller part implemented in hardware, and the upper layers and application logic are software layers. This is convenient for smartphones, because software can be updated and is executed on the phone's processor.

Because of BLE radio modulation, the theoretical upper limit is 1Mbps. In reality, the higher the data rate, the higher is the energy consumption; generally, there is a much lower power efficiency.

Let's suppose a master device is connected with a slave device. The connection interval is the interval between two consecutive connection events, and during connection intervals the

devices are unable to transfer any information. Information is only transferred during connection events. During these events, data is exchanged then devices go back to idle state to save energy.

A connection interval can be set to a value between 7.5 ms and 4 s. The nRF51822 can transmit up to six data packets per connection interval. Each data packet can contain up to 20 bytes of user data (default). So, 133.3 connection events/s * 6 * 20 B = 16000 B/s = 128 kbps.

This is a small number, but it's ok since the design goal of BLE is to save energy. Furthermore, even just 15 KB/s can deplete a coin cell battery, so throughput is generally much smaller.

The basic idea to save energy is to turn the radio off whenever possible, as long as possible, and transmit data in short bursts during connection events - which, still, to save energy, are as separated as they can be.

In general, smaller intervals guarantee responsiveness and higher throughput. Larger intervals save more energy.

In general, connection events are much shorter than sleeping phases.

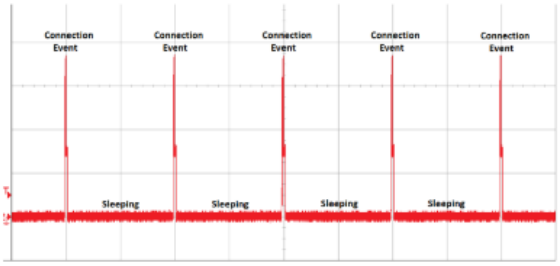


Figure 1- Current Consumption versus Time during a BLE Connection

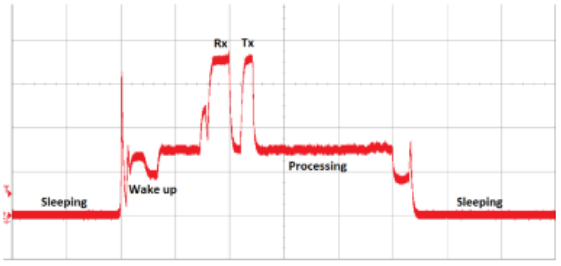


Figure 2- Current Consumption versus Time during a single Connection Event

The transmission power can be configured (it is expressed in dBm). The maximum range is 30 m, the typical range is 2 to 5 meters to save energy. Recent versions of BLE make it possible to dynamically adjust the power, if the RSSI is too high or if the SNR is too low.

Two devices that operate using bluetooth can behave according to a couple of models.
The first possibility is for the two devices to behave as Broadcaster and Observer: the communication between them will be connectionless and unidirectional, only going from the broadcaster to the observer, in the form of small packets. The broadcaster does not know if someone is listening - like a beacon, which sends information. Observers repeatedly scan frequencies to receive any non-connectable advertising packets being broadcasted.

The other possibility is to have a Central/Peripheral model. This is the one that we are going to find when we connect a device to a smartphone or a tablet. The less powerful devices are the peripheral ones. There is a connection established between the two devices, which also agree on some parameters of the connection. This connection is permanent, with a

periodical data exchange of packets between two devices, and starts when the central device discovers the peripheral one.

Data can be transmitted in both directions.

The central device repeatedly scans frequencies for connectable advertising packets, and once a connection is established manages the timing and starts the periodical data exchanges.

The peripheral device periodically sends connectable advertising packets, accepts incoming connections and follows the central's timing and exchanges data with it, when connected.

Topologies can be complex, since:

- A device can act as a central and a peripheral at the same time
- A central can be connected to multiple peripheral nodes
- A peripheral can be connected to multiple central nodes

Still, it can be more energy efficient than an observer/broadcaster system, depending on communication behavior. The delay between connections or data sending can be extended and tuned. Both peers know when the connection events are going to take place, so they can turn off the radio signal for longer.

The physical layer of BLE is organized like this:
It uses the 2.4 GHz ISM band, and communication is divided into 40 channels (37 for connection data, 3 for advertising channels)
The approach is based on frequency hopping: the channel used for transferring data is different at every connection event if there is a channel that is troublesome for some reason, like interference. So, a channel is used and then there is a hop to another one, and the value of the hop is communicated at the beginning of the communication like this:
next_channel = (channel + hop) mod 37

The link layer of BLE is organized like this:
It defines four roles.

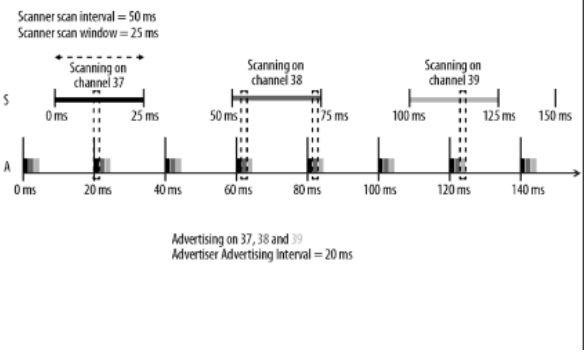
- Advertiser: device sending advertising packets
- Scanner: device scanning for advertising packets
- Master: device that initiates and manages a connection
- Slave: device that accepts a connection request and follows the master's timing

The format of frames and the address of each device is also defined (48 bits, unique)
Actually, two addresses are defined:

- Public device address: equivalent to a fixed, BR/EDR, factory-programmed device address; must be registered with the IEEE, never changes
- Random device address: can either be programmed on the device or dynamically generated at runtime

In the image below we can see an example of the process of advertisement, along with some considerations.

- Advertisements are sent at a fixed rate defined by the advertising interval, which ranges from 20 ms to 10.24 s
- Advertisements are received successfully by the scanner only when they randomly overlap
- *Connectable*: scanner can start a connection
- *Non-connectable*: scanner cannot start a connection (this packet is meant for broadcast only)
- *Scannable*: scanner can send a scan request upon reception of such an advertising packet
- *Non-scannable*: scanner cannot send a scan request upon reception of such an advertising packet

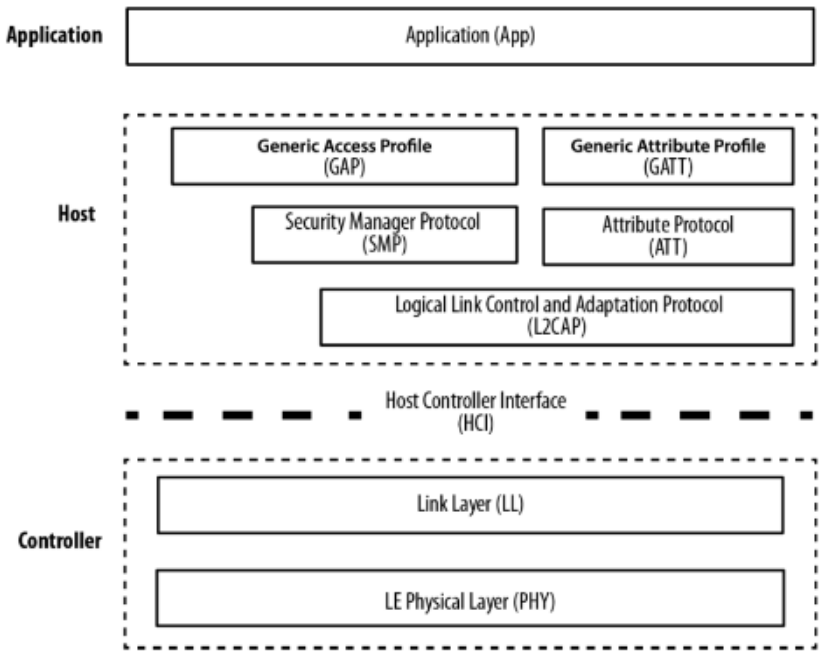


Scannable vs non scannable advertisements: scannable advertisements make it possible to require more information.

A standard advertising packet contains a 31 byte payload, so you can transfer very limited information, including data that describes the broadcaster and its capabilities.

If a 31 byte payload is not enough, it is possible to ask for some more data (scanning response) through a secondary advertising payload. The scanner might be passive (so, listening for advertising packets, and advertiser is never aware of the fact that one or more packets were received by the scanner) or active (in that case, the scanner sends a scan request after receiving an advertising packet, and the advertiser responds with a scan response packet).

After a connection is established, information is usually transferred using data packets that can be sent bidirectionally. The payload of the data is 27 bytes, which in some protocols can be reduced to 20 bytes.



Link layer is reliable: all packets received are checked against a 24-bit CRC, and retransmissions are requested when the error checking detects a transmission failure.

Here is the BLE stack.

After the controller part, with the link layer and the physical layer, we have the L2CAP,

that is basically a layer providing multiplexing and demultiplexing to the upper layers. it also manages possible fragmentation and disassembling of data, and this layer is the one needing four bytes, which means that the effective user payload length is 23 bytes. It interacts with the upper layer, responsible for managing security aspects and implementing the attribute protocol, that is the way information is transferred in the bluetooth world. The idea is to organize the information into a set of attributes that can be read or modified by means of the BLE protocol.

This ATT protocol provides the mechanisms for organizing information as a set of attributes. It defines a couple of roles: servers as the managers of information (which contain data organized in the form of attributes), clients as the ones that need information.

An attribute can be identified by a 16 bit attribute handle, an universally unique identifier, a set of permissions and a value. The attribute handle is an identifier used to access an attribute value and it looks like a row number in a table, whilst the UUID specifies the type and nature of the data contained in the value.
The value depends on the attribute type.

LECTURE 15: 5/5/2022 (Seminar)

From the register: Social Media Analysis and Content Credibility (guest lecturer Stefano Cresci)
….

LECTURE 16: 10/5/2022 (Vecchio)

From the register: The attribute protocol. Security manager, pairing. GAP and GATT. UUID, profiles, series, characteristics. Beacons. Power consumption in BLE. Exposure notification service in contact tracing applications.
….
We were talking about the BLE stack. The details can be found in Lecture 14.

As we already saw, the Attribute Protocol (ATT) is a simple client/server stateless protocol based on attributes presented by a device.

There is a client that requests data from a server, and a server that sends data to the clients. The servers contain data organized in the form of attributes - each attribute has a 16 bit attribute handle, an UUID, a set of permission, a value.

Information always goes from the server to the client.

The handle works as a sort of row number in a table, the attribute type is represented by the UUID and the value depends on the attribute type.

Heart Rate Profile	Handle	Type of attribute (UUID)	Attribute permission	Attribute value
Service Declaration	0x000E	**Service declaration** Standard UUIDservice 0x2800	Read Only, No Authentication, No Authorization	Heart Rate Service 0x180D
Characteristic Declaration	0x000F	**Characteristic declaration** Standard UUIDcharacteristic 0x2803	Read Only, No Authentication, No Authorization	Properties (Notify) Value Handle (0x0010) UUID for Heart Rate Measurement characteristic (0x2A37)
Characteristic Value Declaration	0x0010	**Heart Rate Measurement Characteristic** UUID found in the Characteristic declaration value 0x2A37	Higher layer profile or implementation specific.	Beats Per Minute E.g "167"

The security manager of the BLE stack defines security procedures.

- Pairing, to provide a secure communication channel:
  - a temporary common security encryption key is generated
  - switch to a secure, encrypted link
  - temporary key is not stored and is not reusable in future connections
  - There are different possible pairing procedures:
    - Just works:
      - the devices start communicating in clear text, they exchange temporary keys
      - no security against man in the middle attacks
    - Passkey display:
      - one of the peers displays a six digit passkey on the screen
      - the other side is asked to enter it
      - protects against man in the middle attacks, requires interaction in real life between users
    - Out of band:
      - additional data is transferred without using BLE
      - man in the middle attacks here are also prevented
- Bonding:
  - pairing of the devices
  - generation and exchange of permanent security keys
  - stored in nonvolatile memory, and creating a permanent bond between two devices
  - allows to create a secure link in subsequent connections without having to perform a bonding again
- Encryption re-establishment: after bonding
  - keys generated in bonding are stored on both sides of the connection
  - defines how to use keys in subsequent connections to re-establish a secure, encrypted connection without having to go through pairing and bonding again

Going up on the stack, we find GAP and GATT.

What is GAP? General Attribute Profile, it is a level that defines roles, procedures to broadcast data, discover devices, establish and manage connections, and negotiates security levels. In particular, this layer defines roles (broadcaster observer vs. central peripheral), it defines whether a device is discoverable or not, connectable or not.

What is GATT? Generic Attribute Profile, it defines a model in which the data is organized, and it also defines the operations that can be carried out on the data. There are also notification mechanisms and procedures to discover, read, write, and push data. Data is organized in services.

For example, let's take a heart rate monitor paired with a smartphone.
In the heart rate monitor the GAP role will be peripheral, but the GATT role will be central: it will be a GATT server when the phone requests data from sensors, and a GATT client when it requests configuration information from the smartphone.
On the other hand, in the smartphone, the GAP role will be central.

The GATT client/server roles depend exclusively on the direction in which the data requests and responses flow.
The GAP is always peripheral for the HRM and central for the smartphone.

Let's talk a bit more about UUIDs: an universally unique identifier (UUID) is a 128-bit (16 bytes) number that is guaranteed (or has a high probability) to be globally unique. The only problem is that 16 bytes are a lot if the packets are very small.
So, short formats can be used, but only with UUIDs that are defined in the BLE specification.
To obtain the 128-bit UUID from 16-bit ones:
0000xxxx-0000-1000-8000-00805F9B34FB

Now we can see more in detail how a device can be organized in terms of profiles, services, characteristics.

In particular, any device can have one or more profiles. A profile can define a class of devices, in a way, and services are the ways information is organized in such a device.

In particular, the Bluetooth Special Interest Group defined a standard profile where a profile is defined as a set of services that must be included if the device wants to be compatible with that profile.

Generally, a Profile is a collection of Services that has been defined by either the Bluetooth SIG or by the peripheral vendor. For example, the Heart Rate Profile combines the Heart Rate Service and the Device Information Service

Each service can be recognized by means of a unique numeric ID, the UUID

- 16-bit (for officially adopted BLE Services)
- 128-bit (for not officially adopted services)

E.g.: Heart Rate Service is officially adopted and has a 16-bit UUID equal to 0x180D

Characteristics encapsulate data (for example, Heart rate service has three characteristics, Heart Rate Measurement, Body Sensor Location, Heart Rate Control Point). Characteristics are identified by a 16 or 128 bit UUID, depending on if they are standard or not. The standard ones are defined by the Bluetooth SIG.

This is something like the ATT, the UUID defines how data is encoded.

A common use of BLE is beacons. Beacons broadcast small, even absent, pieces of information, through the broadcaster-sender model. BLE Beacons include iBeacon and Eddystone. They can be used for indoor localization or notify when users are really close to somewhere.
iBeacon:

- a protocol developed by Apple
- smartphones, tablets perform actions when close to an iBeacon

Eddystone:

- made by google, URLs are broadcasted
- at the base of the Physical Web
- automatic notifications even without app
- discontinued at the end of 2018 (many spam notifications)
- now replaced by google beacon

You can compute position by using range and RSSI to compute distance through the power of the signal.
In a beacon, the information that is transmitted is simple. You can configure them to send a maximum and a minimum number, values between 1 and 64k. The idea is to configure the beacon to send numbers to figure your position. For example, you have big shopping centers and beacons for every aisle - the first number is the shop, the second is the aisle.

Now, let's compute the duration of a coin cell battery when powering a TI CC2541, which is a microcontroller unit with a BLE.
The analysis is limited to the BLE stack, without including possible other HW elements, such as accelerometers, etc. Notice that even in high throughput systems, a BLE device is only transmitting for a small percentage of the total time that the device is connected.
If we zoom into a single connection interval, we can see different phases:

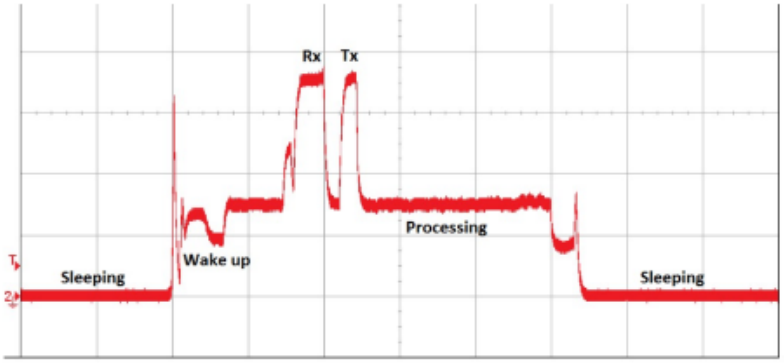
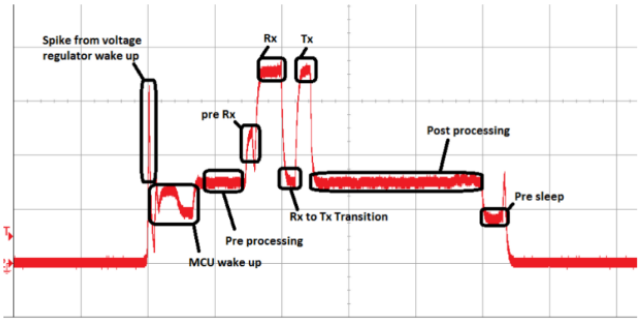


Figure 2- Current Consumption versus Time during a single Connection Event

Receiving data takes the same current as transmitting.

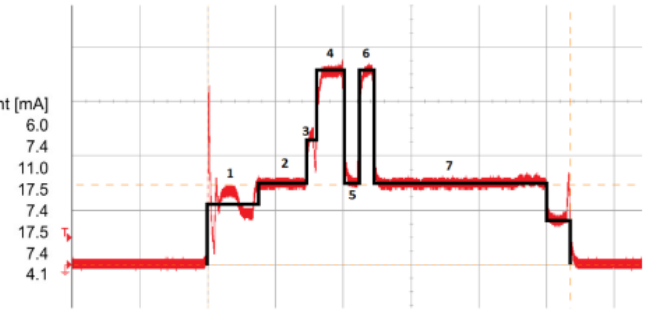
More in detail:

- More in detail
- **MCU wake-up:** upon waking up, the current level drops slightly
- **Pre-processing:** the BLE protocol stack prepares the radio for sending and receiving data
- **Pre-Rx:** the CC2541 radio turns on in preparation of Rx and Tx
- **Rx:** the radio receiver listens for a packet from the master
- **Rx-to-Tx** transition: the receiver stops, and the radio prepares to transmit a packet to the master
- **Tx**: the radio transmits a packet to the master
- **Post-processing:** the BLE protocol stack processes the received packet and sets up the sleep timer in preparation for the next connection event.
- **Pre-Sleep:** the BLE protocol stack prepares to go into sleep mode



it is possible to roughly estimate the duration and the current needed in all these phases.

	Time [µs]	Current [mA]
State 1 (wake-up)	400	6.0
State 2 (pre-processing)	340	7.4
State 3 (pre-Rx)	80	11.0
State 4 (Rx)	190	17.5
State 5 (Rx-to-Tx)	105	7.4
State 6 (Tx)	115	17.5
State 7 (post-processing)	1280	7.4
State 8 (pre-Sleep)	160	4.1



In the end we can compute the average current needed, as the sum of time*current for all states.
Between connection events, the device goes into low power state PM2 where current absorbed can be measured in 1 uA.

We can compute the average current needed for an entire period, that includes a whole connection event plus a whole sleeping state period.

Average current while connected = [(Connection Interval – Total awake time) *(Average sleep current) + (Total awake time)*(Average current during connection event)] / (Connection Interval)

- [(1000 ms – 2.675 ms)*(0.001 mA) + (2.675 ms)*(8.24 mA)] / (1000 ms) = 0.0230 mA

- The average current consumption while the device is in a connected state is approximately 0.023 mA (23 uA)

Since a coin cell battery CR2032 has a capacity of 230mAh, the expected battery life is that value divided by 0,023mA, which is 10k hours, approximately 400 days if always on, with a 1s connection interval and no other sources of consumption.

Now, let's talk about the Google and Apple COVID 19 mechanism which, from contact tracing, saw its name change to "exposure notification" to sound less threatening.
This was a joint effort to enable the use of Bluetooth technology to help governments and health agencies reduce the spread of the virus, and the solution included APIs and OS level technology to assist in enabling exposure notification.

Some of the guidelines:

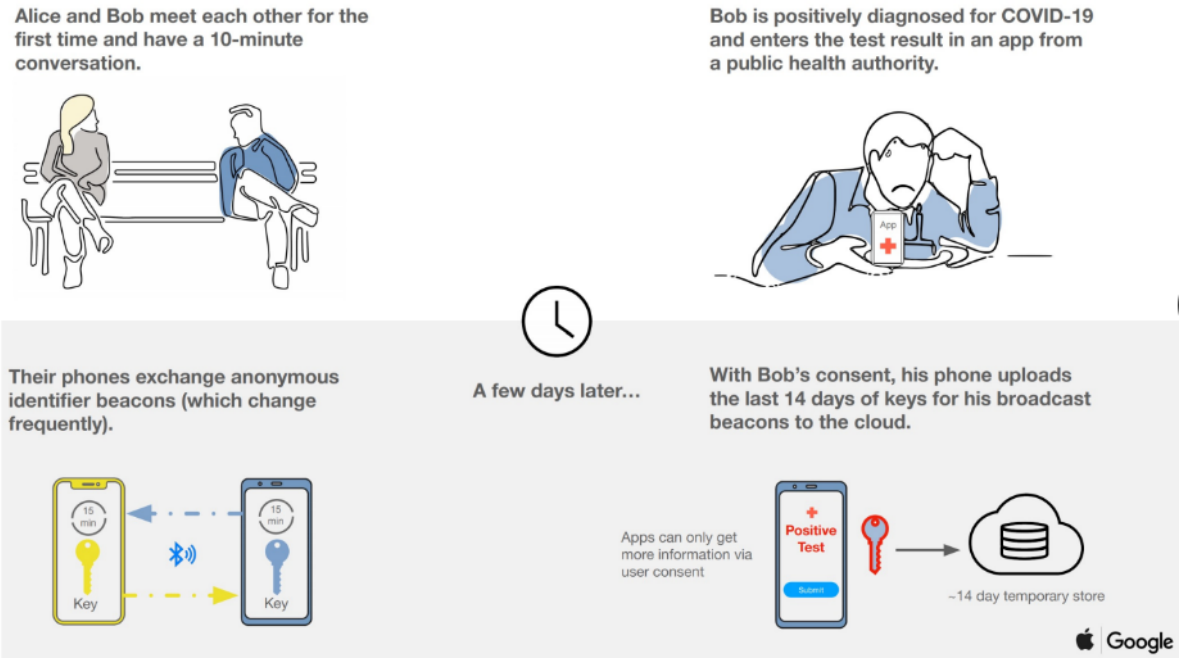
- Explicit consent of the user is required
- It does not collect personally identifiable information or user data
- The list of people that the user was in contact with is not divulged
- People who test positive are not identified to other users, Google, or Apple
- It is only used for contact tracing by public health authorities for COVID 19 pandemic management in authorized applications
- Works on Android and IOS

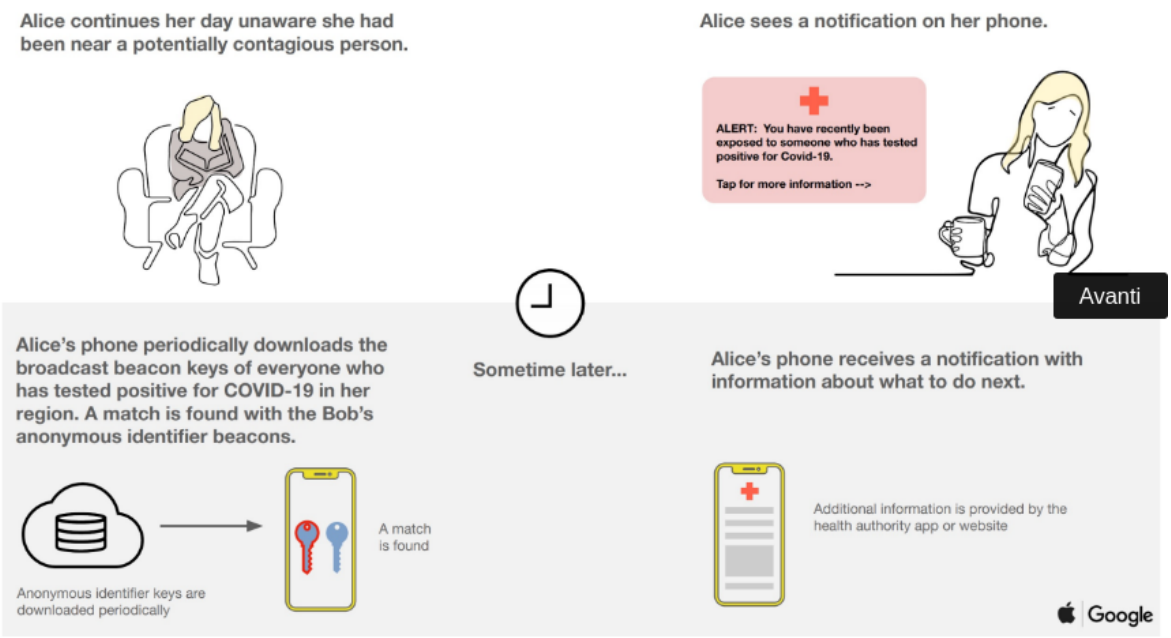
When there is a first contact between two people, the phones exchange anonymous identifier beacons that are randomly generated and change frequently.

If one of them becomes positive, the phone, with their consent, uploads all the keys of the last 14 days to the server.

Phones also periodically download the beacon keys of positive people in the same region, until a match is found.

If a match is found, a notification is sent.





We need a privacy preserving bluetooth protocol to support exposure notification.
Exposure Notification Service is the vehicle for implementing contact tracing and uses the Bluetooth LE (Low Energy) for proximity detection of nearby smartphones. it is a BLE service registered with the Bluetooth SIG with 16-bit UUID 0xFD6F, it is designed to enable proximity sensing of Rolling Proximity Identifier between devices, expressed by a 128 bit string.

The devices broadcast and scan for the ENS by way of its 16 bit service UUID. The service data type with this service UUID contains a 128 bit Rolling Proximity Identifier and Associated Encrypted Metadata, which is useful to understand the distance between sender and receiver.

To be more specific, there are several values.

- Temporary Exposure Key (TEK): a key generated every 24 hours for privacy consideration
- Diagnosis Keys (DK): the subset of Temporary Exposure Keys which are uploaded when the device owner is diagnosed positive for COVID-19
- Rolling Proximity Identifier (RPI): a privacy preserving identifier derived from the Temporary Exposure Key and sent in the bluetooth advertisements
    - It changes every ~15 minutes to prevent wireless tracking of the device
- Associated Encrypted Metadata (AEM): privacy preserving encrypted metadata that shall be used to carry protocol versioning and transmit (Tx) power for better distance approximation
    - it changes about every 15 minutes, at the same cadence as the Rolling Proximity Identifier

There is a Current TEK, given as input to Hashed Message Authentication Codes that produces the RPI key which, together with an integer number that identifies a specific 10-minutes lot in time (starting from Unix epoch) to generate the RPI.

There is another HMAC that turns the Current TEK into an AEM key, which is given as input to a cryptographic procedure that takes as input the RPI, the AEM and the Bluetooth Metadata to give out the AEM.
All of this is given out as the Bluetooth payload.

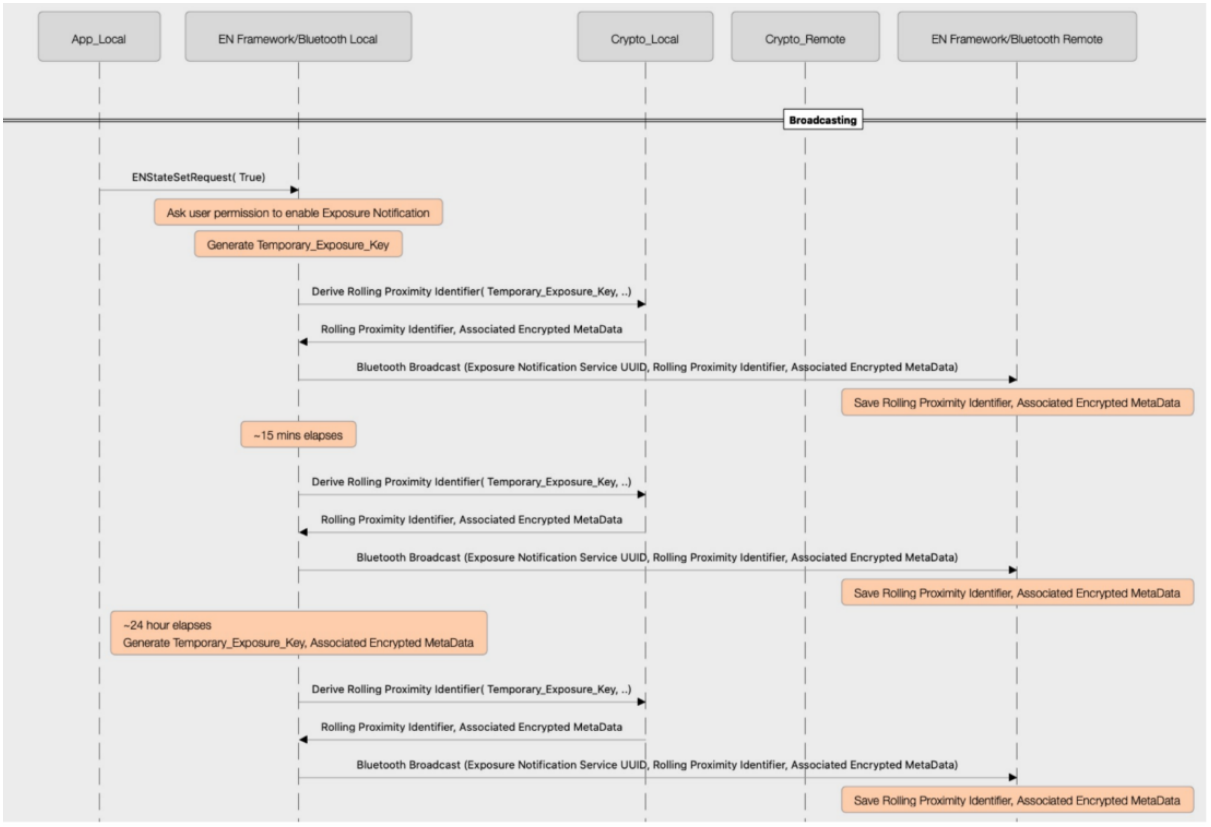
This is the format of every packet which participates in this service.

Flags			Complete 16-bit Service UUID			Service Data - 16 bit UUID				
Length	**Type**	**Flags**	**Length**	**Type**	**Service UUID**	**Length**	**Type**	**Service Data**		
0x02	0x01 (Flag)	0x1A	0x03	0x03 (Complete 16-bit Service UUID)	0xFD6F (Exposure Notification Service)	0x17	0x16 (Service Data - 16 bit UUID)	0xFD6F (Exposure Notification Service)	**16 bytes** **Rolling Proximity Identifier**	**4 bytes** **Associated Encrypted Metadata**

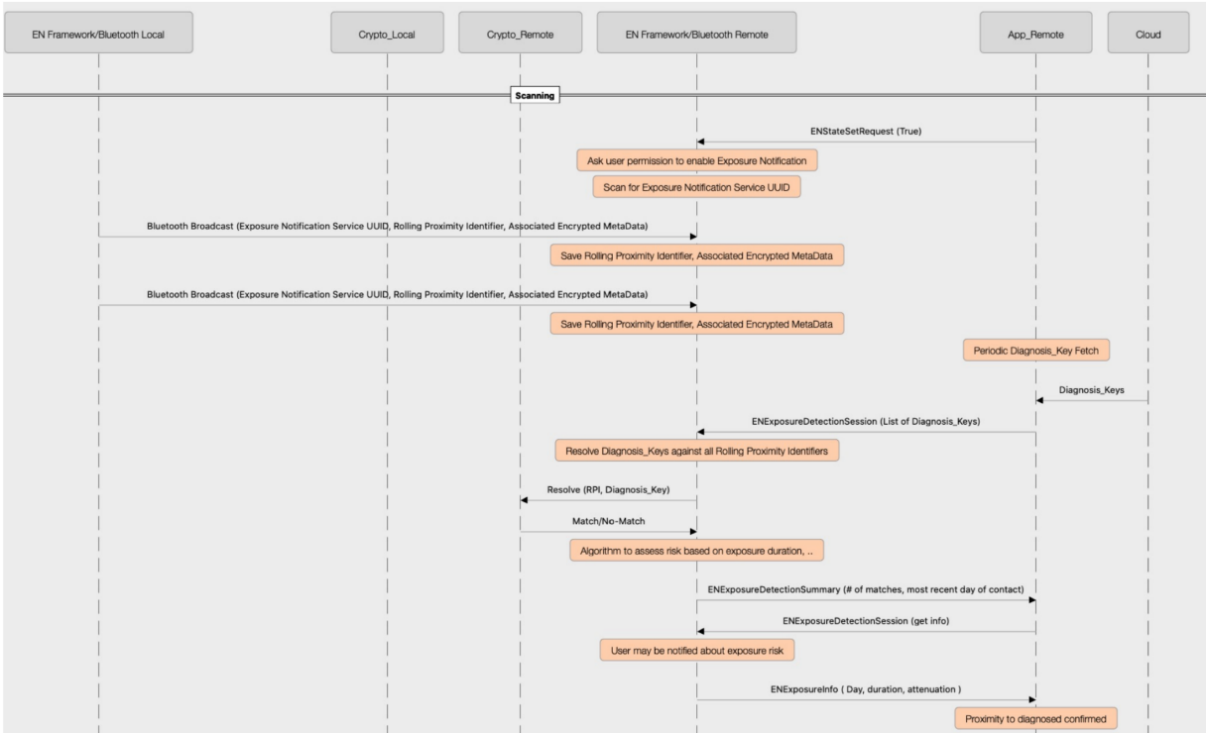
The amount of data that can be transmitted is not so large. What is actually transmitted as a service UUID is a value that indicates that this information is coming from the Exposure Notification Service, which is 0xFD6F.
The payload is composed of 16 bytes of RPI and 4 bytes of AEM, of which the first contains the versioning and the second contains the transmit power level.

The broadcasting interval is recommended to be 200-270 ms.
The advertiser address type should be Random Non-resolvable. On the platforms supporting Bluetooth Random Private Address with randomized rotation timeout interval, the advertiser address rotation period shall be a random value, greater than 10 minutes and less than 20 minutes, to better ensure anonymity.

These are the flows of broadcasting and scanning.



In the transmitter side, an application starts the service and asks permission to the user. If permission is granted, then a new TEK is generated, as well as the RPI, and the information is transmitted with bluetooth advertisement. Receiving devices store RPI and AEM and store them. This is done in 200 to 270 ms, and after 15 minutes a new RPI is generated.
After 24 hours, everything starts with a new TEK.



In scanning, the application starts and asks permission from the user. Everytime a new broadcast is received, RPI and AEM are stored in the persistent storage of the device. Then, periodically, the application downloads the list of DKs of positive people from the cloud. If there is a match between what is in the downloads and what is in the persistent storage, a procedure to indicate what is to be done is started, in relationship with the directives from health authorities.

So, when a user tests positive, a limited set of Temporary Exposure Keys and their respective ENIntervalNumber (describing when their validity started) are uploaded to the Diagnosis Server.
This set of TEKs is limited to the time window in which the user could have been exposing other users (for example, the most recent 14 days).
This subset of keys is referred to as Diagnosis Keys.
The diagnosis server aggregates all users' DKs from whoever tested positive, and distributes them to all user clients that are participating in exposure notification.
Each client periodically fetches the list of new Diagnosis Keys from the Diagnosis Server, which are sets of TEKs with the associated ENIntervalNumber.
Each of the clients can again derive the sequence of Rolling Proximity Identifiers that were broadcast over Bluetooth from users who tested positive: it's the device that derives the 144 (6 values per hour, for 24 hours) RPIs, starting from ENIntervalNumber. The clients match each of the derived identifiers against the sequence they found through Bluetooth scanning.
So, it's my device and only my device that checks whether I've been in contact with positives or not.

The mechanism of the EN service is a responsibility of Apple and Google, but the management of diagnosis keys is in the hands of the health authorities. So, how the cloud mechanisms are implemented, how keys are sent from the app to the cloud, that is all the responsibility of health authorities.

Since they had to distribute this functionality to billions of users in a short time, Google included this functionality into Google Play services, which is updated automatically.

So, the mobile application for COVID tracking must be implemented by authorities.

The applications can calculate the risk for each exposure incident. The principle is: The closer the two people are, the riskier the contact is. The longer the users were together, the riskier. The more recent the event is, the riskier.
So, the application can find a risk level for each parameter, going from 1 to 8. This can be customized by the health authority as well. Each parameter is then multiplied by a weight, and the weighted average is calculated.

LECTURE 17: 11/5/2022 (Seminar)

From the register: Social Media Analysis and Content Credibility: Python crash course. Authentication. Libraries. Python exercise: using Botometer to classify Twitter users. (guest lecturer Serena Tardelli)

Theoretical introduction on how to perform social media analysis.
Outline:
•Introduction to Python
•Introduction to Natural Language Process (NLP)
•Setup Twitter API Account
•Exercise on Twitter Crawling and Social Media Analysis
•Introduction to Bot Detection
•Setup Botometer Account
•Exercise on Social Bot Detection

Python is an interpreted, high level, general purpose programming language. It features a dynamic type system and automatic memory management. It is very useful for data analysis because of the many useful libraries it has.

I will not repeat, in these notes, the introduction on python… everything needed is on the slides and on youtube <3

NLP is a field of artificial intelligence (AI) that enables computers to understand human language by programming computers to process and analyze large amounts of natural language data.
Texts must be processed, and that can be done by following these steps:

1. Tokenization
2. Normalization
3. Filtering
4. Stemming (+destemming)
5. POS tagging
6. NER
7. …

These are preprocessing steps that are necessary before carrying out more complex tasks (e.g., classification).

Tokenization, for example, is a way of separating texts into smaller units, which can be sentences, words etc… and this process might require the use of spaces and punctuations as borders of the tokens.

Normalization means removal of accented characters, as well as the setting of all the characters in lowercase, so that the computer understands that it is the same word.

Filtering is important to focus on the most meaningful words. Articles, conjunction, pronouns, short words and stopwords can be taken out from the dataset.

Stemming is the process of reducing inflected and declined words to their stem - their root form. For example, we can remove suffixes. A few years ago, stemming was a very challenging problem in computer science, but nowadays, there are powerful automated tools that perform this. Destemming is the task of taking the root and forming the base word.

NLP can be applied on Social Media Data in order to extract knowledge from an unorganized bunch of data. We can then visualize the knowledge in a good way.

Let's talk about social media now. Twitter is more public, and text based. it is Public, Short, Popular, Participatory and it offers APIs (endpoints that can be used to understand or build the conversation on Twitter)

Twitter APIs allow us to find and retrieve, engage with, or create a variety of different resources. To use it you must register to twitter and apply for a developer account. After waiting (it might take a while) you will get a Twitter API credential key, which is a long alphanumeric string split in 4 parts.

There's a whole part of exercises I didn't write notes for, sorryyy

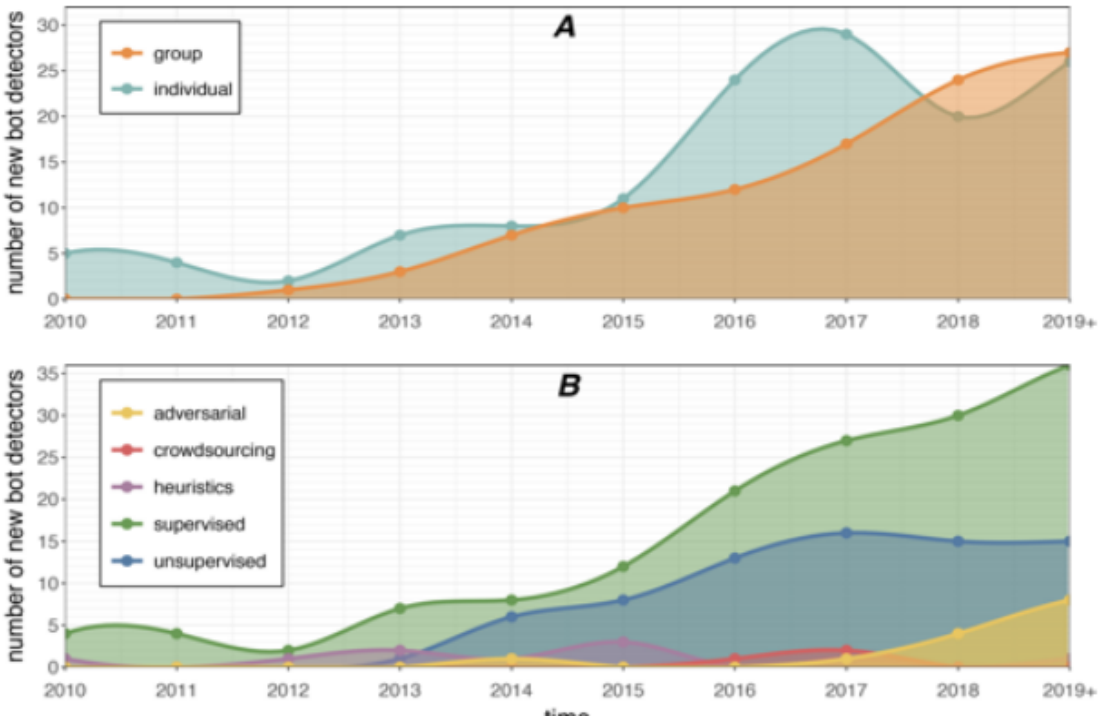
Now, let's talk about Social Bot detection. This is the process of detecting the malicious users on social networks. Social media provides a certain level of anonymity which makes it inevitable for malicious users to be present, since they can do bad without having any real consequences. This can happen in the form of trolls (real humans that make posts with malicious events) or social bots, which are social media accounts that are controlled by a software, in a (more or less) automatic way.

Bot accounts can automatically read, create and post messages, like and reshare content, establish relationships and contact other users. This can have malicious intentions or not. There were some cases in which bots manipulated popular opinion.

Bots, though, can be neutral or benign: for example, for news aggregation and dissemination, support and coordination in emergency situations.

In just a few years, spotting bots has become more and more difficult. Right now, they look like normal users to the human eye.

So, social bot detection can be accomplished with different techniques: at the beginning of this phenomenon most techniques were individual - a single bot account was analyzed to see whether it really was a bot or not.
After some time, bots evolved and masked better, so bot detections evolved into group detection techniques, that is to say, users analyzed in groups to find groups with similar characteristics.
Nowadays, the current trend is to try and detect coordinated behavior, so analysis on the pattern of actions of a group of accounts.



Botometer is a tool that can be used through public APIs and was created in 2015 circa, it is the most widely used social bot classification system. It is based on individual bot detection, it is used in many papers and in particular it is an ensemble of random forest classifiers, each with 100 decision trees and each specialized either on humans or on a specific type of bot: it takes in input more than 1200 features divided in 6 groups (network, user, friends, temporal, content, sentiment).
We can leverage this tool by simply using their APIs, we just need to obtain the key by applying on the rapidapi.com website.

other exercise i didn't take notes on sorry

LECTURE 18: 17/5/2022 (Vecchio)

From the register: Command-line and tools in Android. adb, dumpsys, bugreport. Content of a batterystats file. Battery historian. Measuring the energy consumption through a hardware power monitor. Introduction to energy consumption in mobile devices. Two studies about energy consumption and lessons learned.

Using the command line interface can make some operations more comfortable to be carried out. Android SDK provides many commands available through CLI.

ADB is the Android Debug Bridge, and it allows the programmer to communicate with the device, in particular the client is the executable adbd, and the server speaks with a daemon connected by an usb cable or a wifi/bluetooth connection.
The tool is provided with many functionalities, the adb binary is contained inside the android_sdk/platform-tools/ directory. Some functionalities include installing apps, copying files from and to the device, an unix-like shell, interacting with activities and the package manager, taking screenshots and videos.

example i did not take notes on

In some cases it can be useful to restart the server with "adb kill-server"
To install an app: "adb install file.apk"
There are also push and pull commands to copy and retrieve files from the device.
We can also interact with the activity manager, for example starting an Activity with an Intent
adb shell am start -a "android.intent.action.VIEW" -d "(data identifier uri)"

You can also start a service or send a broadcast intent
adb shell am startservice -n "(name of the service)"
adb shell am broadcast -a "com.example.action MY_ACTION" …

You can also send commands to the Package Manager:
adb shell pm command
for example, the command to list installed packages is "list package", to install "install" and then the path, to uninstall "uninstall" and the name of the package

During install, -g grants all permissions and can be useful for testing.

Dumpsys is a monitoring tool executed on Android devices, it provides information about system services and resource usage of apps.

syntax:
adb shell dumpsys [-t timeout] [–help | -l | – skip services | service [arguments] | -c | -h]

where timeout specifies the timeout period, default value is 10 seconds
l is the list of system services that can provide information (the list is long, they can be selected, disabled, or even used together)
c is the output as csv, machine readable

dumpsys can be used to analyze the energy required by an app, and that is done thanks to “batterystats”. We can reset battery statistics or collect data. Its output is detailed and not well documented. It can be roughly divided in a number of sections:

- Battery history: a time series of power related events, like screen turned on, radio signal etc). This is roughly expressed as an action that changes the power consumption of the device;
- Per-PID stats: amount of time processes hold a wake lock
- Discharge step durations: time between battery percentage charges
- Daily stats: similar to the previous value, but spanning multiple days with charge and discharge events
- Statistics since last charge
    - overall consumption and battery info
    - cellular statistics: status and usage of cellular network
    - wifi statistics
    - bluetooth
    - estimated power use: estimated power consumption of each subsystem and user
    - wake locks
    - statistics by uid: power consumption of the applications identified by the uid

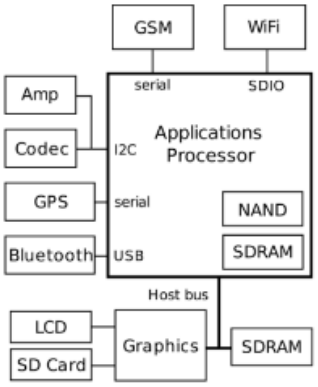
Battery history is composed by a sequence of events, with lines providing the status of the system. There is also some information on the energy policy of the device.
We can also see, among statistics, the capacity of the battery, the brightness of the screen, the amount of discharge, the amount of time when screen was on, the time operating on battery divided in realtime (wallclock time) and uptime (time when the cpu isn't sleeping).

In some cases, when we need a more exploratory approach, we can use the battery historian tool. This tool can be accessed via web browser, and takes as input a trace, and shows the consumption of the battery in a graphical form.
In particular, you can install it using a docker container. Then you have to reset the battery stats:
adb shell dumpsys batterystats –reset
then you have to capture a trace using bugreport
adb bugreport bugreport.zip
then you upload the trace onto the battery historian.

The output is very detailed, and mostly self-explaining.

Now we go on talking about power saving in mobile devices. How is energy spent in a smartphone? It’s difficult to find public studies on how to fix this problem, since limited battery life is one of the main factors adversely affecting the mobile experience of smartphone users. The first step to improve the battery life of a smartphone is to understand how energy is spent, and finding ways to counter that is a professional secret.

The device under test is the Openmoko Neo 2.5G smartphone, whose schematics are freely available.



Component	Specification
SoC	Samsung S3C2442
CPU	ARM 920T @ 400 MHz
RAM	128 MiB SDRAM
Flash	256 MiB NAND
Cellular radio	TI Calypso GSM+GPRS
GPS	u-blox ANTARIS 4
Graphics	Smedia Glamo 3362
LCD	Topploy 480 × 640
SD Card	SanDisk 2 GB
Bluetooth	Delta DFBM-CS320
WiFi	Accton 3236AQ
Audio codec	Wolfson WM8753
Audio amplifier	National Semiconductor LM4853
Power controller	NXP PCF50633
Battery	1200 mAh, 3.7 V Li-Ion

These provide the details of the connections between components. This allows the experimenters to evaluate the power consumption of each module separately, by inserting a resistor in the right point of the main board and evaluating the current.
In particular, for each subsystem, both the supply voltage and the current must be determined. The first one is measured relative to the ground from the component side of the resistors, whilst the current can be computed by sensing a resistor on the power supply rail. Power is then computed as V * i.

The experiments that were done were based on real usage scenarios, like interactive applications and low interactivity applications.

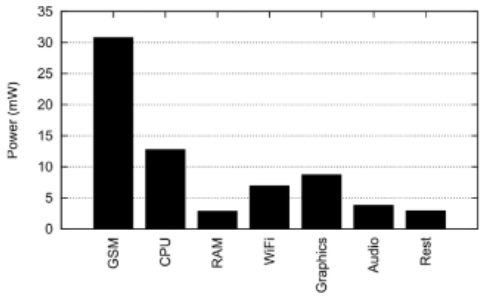
When the activity is analyzed, the baseline state of the device is defined when no applications are running. There are two different states:

- Suspended
    - the processor is in a lower power state, so unable to execute instructions
    - the peripheral components as well
    - the connection to the networks keeps going, but at a low level of activity
    - the status of the devices is stored in the ram
- Idle
    - every component is fully awake, but not doing anything useful

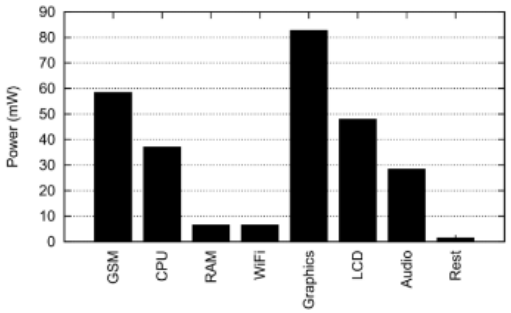
When suspended, the subcomponent that needs more energy is the GSM, but it needs way less.
When the phone is idle, the component that needs more energy is the graphics - followed by GSM, LCD, CPU.

This is important to notice, since moving into the suspended state provides significant saving in terms of energy.

- *Suspeded*:
  - GSM module highest consumption
  - RAM negligible despite mantaining full state
  - total power: 69 mW



- *Idle*:
  - measured with backlight off
  - display-related subsystems highest consumption
  - total power 269 mW



Then, the power needed by the screen is measured: the brightness level is an integer between 1 and 255, the consumption goes from 7.8 mW to 414 mW, and half.scale corresponds to a brightness level of 143, consumption 75mW: the consumption is not linear!
Content displayed on the LCD affected its power consumption: 33.1mW for a completely white screen, and 74.2mW for a black screen. Of course, this depends on the technology used for the screen.

The audio playback accounted for a total power of 320 mW, circa 12% of energy consumption, with the screen turned off. Maintaining a connection to the GSM network requires significant power, whilst if the MP3 file is loaded from the SD card, the cost of doing so is negligible.

Video playback with no backlight accounts for a total power of 453 mW; the biggest single consumer of power, other than the backlight, is the CPU - the display subsystems still account for a large fraction. The backlight alone can go up to 400mW alone!

A phone call consumes a total power of 1054 mW, more than 800 due to the GSM.

The amount of power needed for web browsing is 352mW if we are using WIFI, 429 for GPRS - which consumes more power than WIFI by a factor of 2,5.

There are some lessons that we can learn from this study.
For example, one of the largest factors of power consumption is screen lighting - that is why some smartphones implement aggressive backlight dimming and inclusion of ambient light and proximity sensors in mobile devices to assist with selecting appropriate brightness.

Moreover, even just maintaining a connection with the network consumes a significant fraction of power. In a call, GSM consumes more than 800mW, the single largest power drain.

There is a static contribution to system power that is still relevant - this motivates us to shut down unused components and disable their power supplies when possible.

This study was hardware based and the measurements were accurate, but the scenarios were not so realistic.

We will now look at another study where power consumption is measured according to a less accurate method, but in a more realistic scenario.

The basic idea is that we involve a lot of users to obtain a realistic model of how a smartphone is used. Now we are using a software based approach similar to the one we consider when analyzing battery stats.
In particular, they defined a number of Finite State Machines to model the consumption of the different subsystems, obtained a trace of how the smartphone was used and then they drove these FSM with the trace collected from real users. In particular, they wanted to do this without requiring evolved smartphones. The application was a standard one that could be installed by approximately 1500 users on Samsung s3 and s4.

So, let's consider the main components of the smartphone, then consider three baseline activities, like wifi beaconing, cellular paging, and the amount of power needed for the minimum operating functionality of the device.

In particular, the model of the CPU was based on the operative frequency and utilization level. So, the power required by the CPU is equal to a baseline when a number of cores is used. This method was measured in a lab with the same devices. We add to this term the utilization level of the different cores of the CPU, and this is the additional power needed by a given core when operating at a frequency.

Hardware component power draw	Model trigger
CPU	frequency + utilization
GPU	frequency + utilization
Screen	brightness level
WiFi	FSM + signal strength
3G/LTE	FSM + signal strength
WiFi beacon	WiFi status
Cellular Paging	cellular status
SOC Suspension	constant

$$P_{CPU} = P_{B,N_c} + \sum_i^{N_c} u_i \cdot P_\Delta(f_i)$$

$P_{B,Nc}$ is the baseline CPU power with Nc enabled cores
$P_{\Delta(fi)}$ is the power increment of core *i* at frequency *fi*, and *ui* is its utilization.

Measured in lab, always the same devices

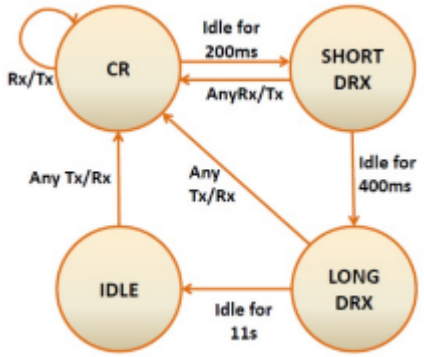
At runtime, the energy of each app was estimated using the logged CPU frequency and utilization

So, this model describes the amount of power needed by the CPU when a number of cores are active at a certain frequency according to the utilization level. This can be estimated only for a specific CPU by estimating coefficients during lab sessions.
The same approach has been adopted for the GPU.

What about the screen? The consumption has been measured at different levels in the lab and the screen brightness level was logged without considering the displayed content.

So, this was the model of the cellular interface, which can be in one of these four states:

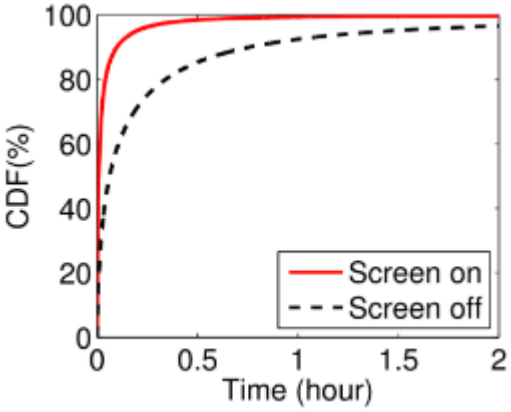
1. Continuous reception, the state requiring the maximum amount of power, in the order of 1000mW. here it can receive and transmit data. If it remains idle for 200 ms, it goes to
2. Short DRX, which is Short Discontinuous Reception, the overall energy consumption is lower compared to the first state. In particular, here the interface remains turned off most of the time, and it turns on for very short periods to check for incoming



messages. if there is an incoming message or something to be transmitted, the interface goes back to CR. Otherwise, after 400 ms it goes to

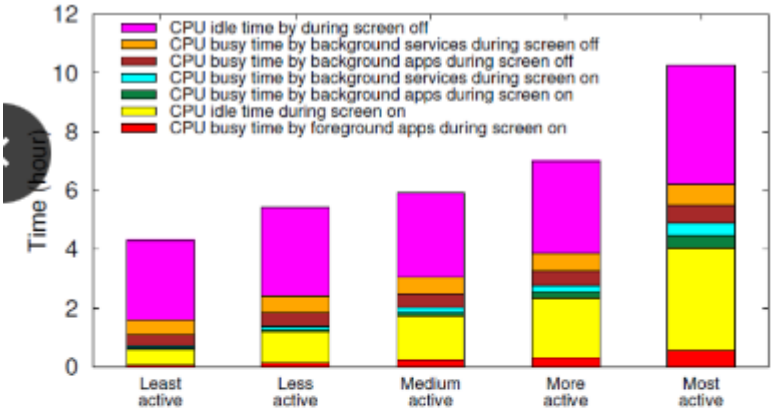
3. Long DRX, where the interface wakes up for checking if there is something to be received or transmitted. If that is the case, it goes back to CR. Otherwise, after 11 seconds, it becomes
4. Idle: lowest energy settings.

The less energy spent, the more time it takes for the system to react to any Tx/Rx.

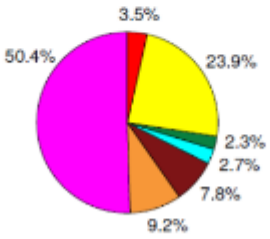


Wifi is modeled in a similar way to LTE, it can depend on how frequent transmission is and how long silence is.

The baseline associated with wifi beaconing and cellular paging: in that case the interface is in the idle state, and there is a spike every 1,28 seconds. The average current needed was quite low. They made something similar for WIFI to estimate the energy needed to look for access points.



(a) Average daily CPU time breakdown of 5 groups of the 1520 users.



(b) Daily CPU time percentage breakdown, average over all users.

Figure 5: Daily CPU time breakdown.

They also collected some statistics from real users and the empirical cumulative distribution function of the amount of time the screen was on or off.

They computed the CDF of these periods of times, and we can see that 80% of the time, the screen remains on for a few seconds. Also, 80% of the time the time between screens-on is 15 minutes. Obviously, we can tune the system to obtain a better energy consumption.

They then divided users into 5 groups by total screen-time, to see what the CPU was doing. What was observed was that most of the

applications are not using the application processor. Almost 24% of the time, the CPU was idle but the screen was on!

The activity could be something like: the user is reading a mail, browsing the web, etc.

We can also observe that 50% of the time the screen is off and the CPU is idle. This should ideally be close to zero, but there are many small activities that prevent the processor from going to the deepest sleep state.

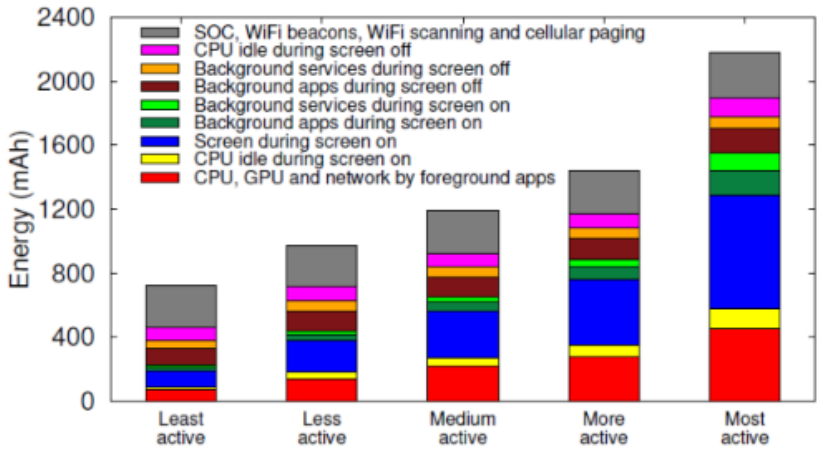
LECTURE 19: 18/5/2022 (Vecchio)

From the register: Energy consumption in mobile devices. Heterogeneous computing. Dynamic voltage and frequency scaling. Scheduling in heterogeneous computing architectures. Mechanisms at the OS level (Android): doze mode, app standby. Techniques for efficient communication. Application-level mechanisms for reducing energy consumption.

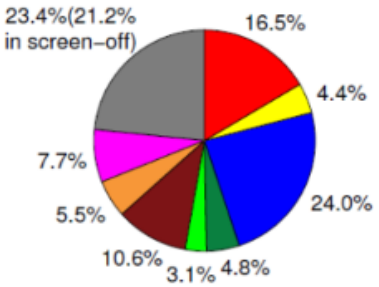
….

Let's keep talking about how energy is spent in a smartphone.

These images make it clear how the different components spend phone energy. The amount of battery used for cellular paging is 12,5% compared to WiFi beacon which is 2,3%, and the total energy drain over cellular and over WiFi are 11,7% and 1,3% respectively.



(a) Average daily energy drain breakdown of 5 groups of the 1520 users.



(b) Daily energy percentage breakdown, averaged over all users.

These results are interesting because they were collected from many different individual usages of a smartphone.

There are some lessons to be learned from the two studies.

First of all, the screen is responsible for a large fraction of energy consumption, and this motivates the automatic light adjusting.

Second, WiFi is more energy efficient than cellular technologies.

Moreover, the CPU is idle for quite some time because of a constant light computational load.

Finally, background activities consume quite a large amount of energy.

In particular, focusing on the CPU, the computational load is generally light, with some spikes and some occasions when high computational power is needed.
A high performance CPU makes a user happier, but it is a waste of energy most of the time.
A low performance CPU reduces power consumption, but along with user satisfaction.

This is why almost every smartphone adopts heterogeneous multiprocessing.
for 20 years, to improve the performance of a processor, the common reasoning was to increment operational frequency. Then, the maximum frequency that was reasonably usable in a processor was reached, and the approach was to add more cores and achieve major performance by means of increased parallelism, where all processors and cores are equal. This is called a Symmetric MultiProcessor.

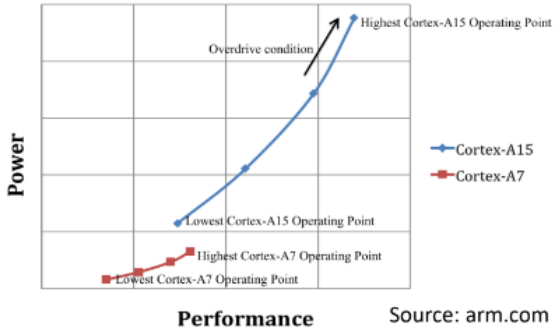
In Heterogeneous MultiProcessing, there are differences in power and in efficiency through processors and cores. So, we get superior performance in terms of energy saving.

ARM's big.LITTLE is an architecture that combines high performance and smaller CPUs in one system; software dynamically moves to the right size processor for the required performance.

- big: A75, A15, …
- LITTLE: A55, A7, …

- A15 vs A7:
  - A15: out-of-order execution, triple issue, pipeline 15-24 stages
  - A7: in-order execution, double issue, pipeline 8-10 stages
  - Performance: A15 vs A7 ~2x
  - Energy efficiency: A7 vs A15 ~3x
  - Same instruction set, same registers

Both use Dynamic Voltage and Frequency Scaling (DVFS)

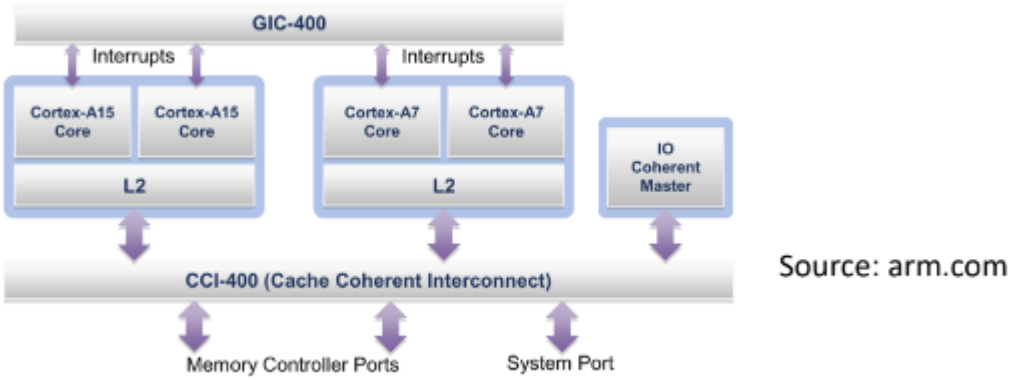


Source: arm.com

The idea behind Dynamic Voltage and Frequency Scaling:

It is an energy saving technique that exploits the linear relationship between power consumption and frequency, and the quadratic relationship between power consumption and voltage.
This relationship is given as P = C * V^2 * f
where P is the power, C is the switching capacitance of the logic circuit, V is the operational voltage, f is the operational frequency.

Any higher frequency is going to require more power, whilst lower frequencies can operate at lower voltages.

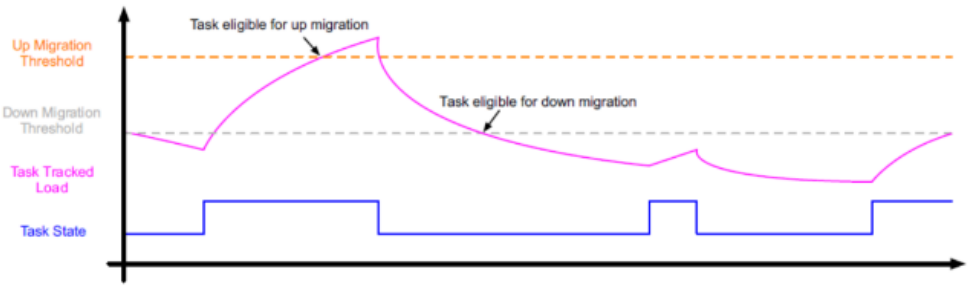


Two big cores and two little cores share a second level cache and there is also a system that redirects the interruptions to the correct core.
There is also an interconnection of the caches to avoid writing in memory.
This is particularly useful when migrating processes between cores.

Three main task migration models are possible:

- Clustered switching
    - Clusters of big and LITTLE core have the same size and they are all united;
    - Just a single cluster can be active at any time;
    - When the load becomes high, all tasks are moved to the big cluster, and vice versa;
    - The inactive cluster is powered off;
    - Not all cores are used;
    - It is simple but suboptimal, in terms of both energy and computing power (4 cores are always inactive and not all processes need the same type of CPU)
- CPU migration (in kernel switching)
    - We have virtual cores, made with a big and a LITTLE core;
    - When the load on a CPU becomes high, that CPU moves the tasks on a big core and vice versa;
    - The inactive real core is powered off;
    - Not all cores are used;
    - Simple but suboptimal both in terms of energy and computing power (4 cores are always inactive, but it is a bit better than the other one energy wise)
- Heterogeneous MultiProcessing (global task scheduling)
    - Numbers of big and LITTLE cores might be different
    - When the load in a LITTLE becomes high, the task gets moved to a big core and vice versa
    - Inactive cores are powered off;
    - all cores may be in use at the same time
    - The scheduling decisions are more complex.

The scheduler, in this last case, must track the activity level of each task, since it is a task that gets moved. For this, we get a counter and we set two tresholds.



If we consider an inbound and outbound processor, where the outbound is the one that is currently running the task and should be turned off as soon as possible.

The first step is for the outbound processor to send a signal to the inbound one, to make it power on and reset. The inbound processor will then invalidate its cache and enable snooping, whilst the Outbound processor will keep operating normally.
When the inbound processor is ready, it sends a signal.
The migration can now take place. The outbound processor saves the task's state and migrates it to the inbound processor, where the task will resume with its normal operation.
The state that is saved on the outbound processor is snooped and restored on the inbound processor rather than going via main memory, to save time since we have a common cache.
The level-2 cache of the outbound processor remains powered up after a task migration to improve the cache warming up time of the inbound processor through snooping of data values.
When this process is finished, the L2 cache is cleaned, snooping is disabled and the outbound processor is powered down.

To manage a system with different computational power and energy requirements available, the system has to employ a scheduler that is aware of the characteristics of the single cores.
The default scheduler in a linux system is the Completely Fair Scheduler, which has been designed to maximize throughput and performance of interactive applications.
This is not responsible for managing the power state and operational frequency of the CPU.
There are two components of the kernel that take care of this: cpuidle and cpufreq.

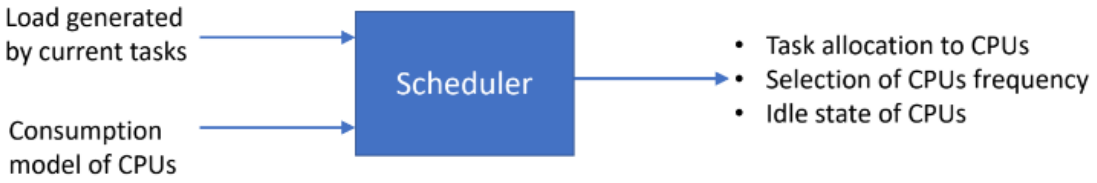
cpuidle moves the CPU into one of the available low power C states (there are several possible ones). The lower the power the longer the time to return operational; cpuidle is called when CPU is idle to select the C state to enter.

cpufreq selects the operational frequency of the CPU. In general there is the ondemand governor, which defines the policy according to which the operational frequency of the CPU must be changed, and keeps it between a lower and an upper threshold frequency.

The problem is that these three components have different views of the system. In particular, if we are interested in making power aware scheduling decisions, this can be done more easily by the scheduler.

This is why in Android a new scheduler was introduced: the Energy Aware Scheduler, or EAS.
The idea is to track the load of every task and then make a decision that considers also the power requirements of the different elements of a CPU.

The changes introduced by the EAS are effective only when there is spare capacity. If the load is high, there is nothing to save and behavior returns to the standard CFS.



The consumption model is basically a file that describes the requirements of the different subsystems of the processor, and it takes into account:

- topology of the different subcomponents of the CPU
- power consumption for each supported operating point
- power consumption for each c-state
- wake-up energy costs for each c-state
- data for CPUs and clusters

To track the load of every task, the scheduler uses a mechanism called Window Assisted Load Tracking (WALT): the time is divided in windows of 20 ms, the load of every task is evaluated for each window. Then, the predicted load for each task will be calculated as the maximum load during the last windows and the average of the last N windows (where N is around 5).
If we use the maximum, the system reacts more quickly to surges in demand.

The other system, the Per Entity Load Tracking (PELT) functionality, is similar but calculates the load only with averages, where older values count less in an inverse quadratic fashion.

Windows do not exist when a task is blocked: a heavy task exiting from sleep is immediately classified as demanding. This is because we want to act as fast as we can.

So, the scheduler is aware of the load of every single task. When a task wakes up, it will need a processor for its execution and the scheduler will allocate the task according to the current load of the different cores, and also on their power requirements.

If, for example, we have to insert a somewhat big waking task and we must choose between a very powerful CPU vs. an adequately powerful CPU with a lower current capacity, we have to choose:

1. Do we raise the operating point of the smaller ones?
2. do we use a more performing cpu, but not exploiting it entirely?

EAS will probably follow the first path. The small additional energy costs of increasing the operating point of the smaller CPUs is not significant compared with the efficiency of running the task on the more energy saving CPU.

How can we save power at a higher level?
Android gives us some power saving mechanisms.
The first one is doze mode, which is a mode activated automatically when the device is not connected to a power source, and it is left stationary for a long period of time. Background CPU and network activity are deferred.
The second one is App standby: Android limits the resources available to all the apps, depending on how frequently and recently they have been used.

Doze mode:

before doze mode, there are many small CPU bursts. During Doze, Network access is suspended, WiFi scan is not performed, alarms and periodic activities are deferred and wake locs are ignored. The device exits doze mode when moved, connected to a charger, or when the screen is turned on.

This is, instead, how the App standby mechanism works.

Bucket	Jobs	Alarms	Network	FCM
Active:	No restriction	No restriction	No restriction	No restriction
Working set:	Deferred up to 2 hours	Deferred up to 6 minutes	No restriction	No restriction
Frequent:	Deferred up to 8 hours	Deferred up to 30 minutes	No restriction	High priority: 10/day
Rare:	Deferred up to 24 hours	Deferred up to 2 hours	Deferred up to 24 hours	High priority: 5/day

The idea is to classify the applications in different buckets, from currently used to regularly used to often used (but not daily) ending with the non frequently used ones. When the device is not connected to a power supply, the restrictions are imposed to apps depending on their bucket.

Related concepts are:

- Works/Jobs: operations to be executed when specific conditions are met (e.g. network is available, device is charging, timeout)
- Alarms: mechanisms useful to perform time-based operations even when an app is not running.
- Firebase Cloud Messaging: a mechanism useful to send messages to android smartphones (e.g. new data on a server is available)

Let's now focus on the energy consumption caused by communications - we know that this is important since cellular communication is one of the most intensive, speaking in terms of energy usage. In particular, every cellular interface can be modeled as a Finite State Machine.

In the example of a 3G radio, the energy depends on its state:

1. Full power is the state when the connection is active, and data can be sent or received - it needs a high power, and if it stays idle for 5 seconds it goes to the second state
2. Low power is the state where data can't be transferred, power consumption is around 50%, and if required can go back to the first state with 1.5 seconds of latency - if that doesn't happen for 12 seconds, the radio goes to the third state
3. Stand by, where data cannot be transferred, power consumption is minimal, and it takes 2 seconds of latency to go back to full power.

Let's now suppose that our application is going to transfer data:

- Suppose an app transfer data for 1 second, 3 times per minute

- Same app that takes care of aggregating transfers

In the first case, we did not take the transitions into account.
The second way of managing the network is more energy efficient.

Generally, we can say that transferring data all together but less frequently is more energy efficient. This can be easy when the data transfer depends on the app logic (sending logs or synchronizing a database), but it's more difficult when the transfer depends on the user's actions.
In this scenario, keeping the application responsive is more important than reducing the power consumption.

There are cases in which we do have viable strategies - for example, in a blog or in a social network app, the implementation can download chunks of entries all together and show them when the user scrolls, in order to find a tradeoff between aggregation and downloading unnecessary content. Another example might be a video streaming application in which the video is downloaded in chunks at regular intervals, in order to prefetch the portion of the video that the user is probably about to watch.

To this purpose, we can define the WorkManager Class, and define some constraints that define when some operations have to be carried out. In this example, we first define a set of constraints by means of a HelperBuilder class, this chain of methods and the build method at the end. These generate a constraint object. There are several options that can be selected in this case.
The idea is to do this operation only when the network type is unmetered, so it could be a WiFi connection instead of a cellular one.
Also, we want to carry out this operation only if the battery level is not low.
So, this is the set of constraints. Then we have to create a WorkRequest, this is done using a Builder class. This final build method creates the action on request.
We want to define a periodic operation, in this case, and the code of the operation we want to perform is defined in the DownloadSetOfPostsWorker class. We want to carry out this operation every 15 minutes and, in case the operation is not successful, we can define a back off criteria to try again the operation. In this case, we want to retry after 30 seconds, linearly.
It is also possible to provide data to the worker - this is done by means of a Data object that is built according to this technique. We are just providing a set of URLs, it's a key value pair that we are providing to the worker itself.
Then, the work to be done is actually provided to the system by means of this final line of code.

```
Constraints constraints = new Constraints.Builder()
    .setRequiredNetworkType(NetworkType.UNMETERED)
    .setRequiresBatteryNotLow(true)
    .build();
WorkRequest request = new
PeriodicWorkRequest.Builder(DownloadSetOfPostsWorker.class, 15, TimeUnit.MINUTES)
    .setBackoffCriteria(BackoffPolicy.LINEAR, 30L, TimeUnit.SECONDS)
    .setInputData(
        new Data.Builder()
            .putString("URLS_TO_BE_DOWNLOADED", "http://...")
            .build())
    .build();
WorkManager.getInstance(this).enqueue(request);
```

In the second part of the code we have the definition of the DownloadSetOfPostsWorker, to extend the system Worker class. The code to be executed is actually provided by this doWork method.
The first line of code retrieves the input data from the worker that is, in our case, the set of urls to be downloaded.
If the input data is missing, this result.failure() alerts the system. Otherwise we start the work, defined in a separated method, and return the success.

In this case, downloadASetOfPosts(inputData) is an empty method, and this is where we should put the actual code to download the data from the network. Right now we just need to

understand the fact that we can ask the system to start a method only when different constraints are met.

```
public class DownloadSetOfPostsWorker extends Worker {
  public DownloadSetOfPostsWorker(@NonNull Context context,
                                  @NonNull WorkerParameters workerParams) {
    super(context, workerParams);
  }

  public Result doWork() {
    String inputData = getInputData().getString("URLS_TO_BE_DOWNLOADED");
    if(inputData == null) {
      return Result.failure();
    }
    // Do the work in other method
    downloadASetOfPosts(inputData);

    // Indicate whether the work finished successfully with the Result
    return Result.success();
    /* or Result.retry(): The work failed and should be retried. */
  }
  private void downloadASetOfPosts(String s) {
    // Actual network operations ...
    Log.i("NETOPT", "inputData: " + s);
  }
}
```

Another possibility is to change some details about the network operations, depending on the type of connections - in particular, the amount of prefetch we want to adopt could be different from WiFi and cellular connection, and in the case of a cellular connection it could be different for fast ones and slow ones.

In this case, the idea is to register a network callback, to be notified by the system when there is a change in network connection, to see if there was a change in the type of network, and operate accordingly.

To do this we must obtain first a reference to the connectivity manager, then we are going to register a network callback that is actually an object providing an implementation for this network callback interface. This is an internal interface of the connectivity manager class. This interface includes a method, onCapabilitiesChanged, that is going to be executed by the system when a change is detected to understand what changed. We are going to use the TelephonyManager class, again, obtained by the system service method.
Then there are some methods that can be applied on the NetworkCapabilites' argument, to understand the nature of the network.
We then set a different prefetchSize for each different type of network.

```
private int prefetchSize = 10;
private boolean wifiAvailable = false;
private boolean cellularAvailable = false;

public void go2(View v) {
  if (ActivityCompat.checkSelfPermission(this, Manifest.permission.READ_PHONE_STATE) !=
     PackageManager.PERMISSION_GRANTED) {
     ActivityCompat.requestPermissions(this, new String[]{Manifest.permission.READ_PHONE_STATE}, 1);
     return;
  }
  ConnectivityManager cm =
        (ConnectivityManager) getSystemService(Context.CONNECTIVITY_SERVICE);
  cm.registerDefaultNetworkCallback(networkCallback);
}

   private ConnectivityManager.NetworkCallback networkCallback = new ConnectivityManager.NetworkCallback() {
     @Override
     public void onCapabilitiesChanged(
          @NonNull Network network,
          @NonNull NetworkCapabilities networkCapabilities
     ) {
        super.onCapabilitiesChanged(network, networkCapabilities);
        TelephonyManager tm =
             (TelephonyManager) getSystemService(Context.TELEPHONY_SERVICE);

        cellularAvailable = networkCapabilities
             .hasTransport(NetworkCapabilities.TRANSPORT_CELLULAR);
        wifiAvailable = networkCapabilities
             .hasTransport(NetworkCapabilities.TRANSPORT_WIFI);
        if(wifiAvailable) {
           prefetchSize = 10;
           return;
        }

                int nt = tm.getDataNetworkType();
                switch (nt) {
                  case TelephonyManager.NETWORK_TYPE_LTE:
                     prefetchSize = 8;
                     break;
                  case TelephonyManager.NETWORK_TYPE_EDGE:
                     prefetchSize = 4;
                     break;
                  default:
                     prefetchSize = 1;
                     break;
                }

             }
           };
```

There is another strategy to reduce the energy consumption due to communication is to not communicate at all, and cache information. Sometimes using the network requires more energy than reading from local storage. We can build our app specific mechanism or we can use the OkHttp library, which uses caching.

In the example, we create a cache for the OkHttp library, we have the size in megabytes, the directory where data gets stored, and then we simply create the client and use it for placing the requests.

```
private OkHttpClient client;
private Cache cc;

public void go3(View v) {
  if(client == null) {
    Log.i("NETOPT", "Creating client...");
    File cacheDir = getCacheDir();
    long cacheSize = 10L * 1024L * 1024L; // 10 MiB
    cc = new Cache(cacheDir, cacheSize);
    client = new OkHttpClient.Builder()
        .cache(cc)
        .build();
  }

  Request request = new Request.Builder()
        .url("https://publicobject.com/helloworld.txt")
        .build();
```

In this case, we have the request for an URL that we then send to the client.

The approach is asynchronous in terms of execution flow. We cannot use the main thread of an app to carry out our own operations, if they are too long - and that includes using the network.

So, to move the execution of the network request out of the main thread of the application, we can use asynchronous execution provided by the library. So, if the request fails the request will be executed again in the future; otherwise, the onResponse method will be executed.
In this case we just extract the body from the response and then log some information on the log.

```
client.newCall(request).enqueue(new Callback() {   // Remember, no network operations in main thread
    @Override
    public void onFailure(Call call, IOException e) {
      Log.e("NETOPT failure", e.getMessage());
    }
    @Override
    public void onResponse(Call call, Response response) {
      try {
        Log.i("NETOPT", "requests: " + cc.requestCount() + ", hit: " + cc.hitCount() + " , net: " + cc.networkCount());
        ResponseBody responseBody = response.body();
        if (!response.isSuccessful()) {
          Log.e("NETOPT", "Response: " + response);
        }
        Log.i("NETOPT", responseBody.string());
      } catch (Exception e) {
        Log.e("NETOPT", e.getMessage());
      }
    }
  });
}
```

LECTURE 20: 24/5/2022 (Avvenuti)

From the register: Humans as Sensors: Data Collection, Data Structuring, Sources Relationship, Estimation Problem. Example. Methodology. Validation Method. Precision Evaluation. Observations and Limitations.

When we want to use observations from a social network, first we have to choose and crawl the social network. In this course we crawl tweets, and we only consider observations that are not subjective. When we collect this data, then, we need to structure it, we should transform a post into something usable, and one idea is tokenizing it.
After having transformed our posts into usable data, we can cluster it by finding a distance between posts and grouping the closer ones together. This cluster is a claim. This makes us able to build the Source Claim Graph. (SC)
Then, we have two other problems: the first one is that we must know which claims are actually true, since humans are not always reliable.
The second one is that humans propagate information among themselves.
To help with this, we can assume the existence of a latent social information (SD) dissemination graph, that estimates how information can propagate from one person to the other. This can be done in 4 ways: epidemic cascade network; follower-followee network; retweeting network; combined network.

Now, we have to estimate the correctness of the claims. From the mathematical point of view, even if we do not have a ground truth we might be able to build one. For example, if the observations are talking about a hurricane, it's possible that if it's true, the newspapers will soon follow. Those news outlets can be used as a ground truth.
Another point is that claims are either true or false. This reduces the problem to one with a boolean answer, so we can calculate the probability that a given boolean variable is true or false.
So, how can we estimate a claim?
There are many different approaches.

Voting determines that the claim with more support is the most believable.
This is not optimal: different sources have different degrees of reliability and their vote should weigh more (but how much? We cannot know!), and sources sometimes are not independent and simply repeat what they hear from others.

We can try to use classical probabilistic techniques. We observe an unknown phenomenon and the way the phenomenon manifests itself is given by a number of different observations that we have to put together in order to let some truth emerge.
Likelihood estimation lets us compute the probability of correctness of claims taking into account unknown source reliability and uncertain provenance. So, given SC and SD, we can compute the conditional probability that a certain claim C is true, for each claim.

$$\forall j,\ 1 \le j \le N : P(Cj = 1 \mid SC, SD)$$

We do this because we know that SC and SD are true. Still, it is very hard to approach.

Maximum Likelihood Estimation is an optimization approach. In order to do this we have to introduce some definitions first.
Let m be the total number of sources from which we have data. We can describe each source Si with i that goes from 1 to m, by two parameters unknown in advance:

- The odds of true positives: ai = P(SiCj = 1|Cj = 1) - this gives us the reliability of a given source

- The odds of false positives: bi = P(SiCj = 1|Cj = 0)

Now, let d be the unknown expected ratio of correct claims in the system d = P(Cj = 1)
Let Θ = [a1… am, b1… bm, d] be the vector of the unknown parameters above.
Since we do not know the value of all these variables we want to build a Theta vector that better allows us to construct the SC graph that is the only evidence that we have on the phenomenon. Generally we use iterative algorithms that start from a certain value for Theta and give us a converged Theta that is the solution to our problem.
Formally we can describe the ML Estimator as follows: we have to find the value of the unknown vector Theta that maximizes the probability of observation of SC given the social network SD. $P(SC|SD, \Theta)$.

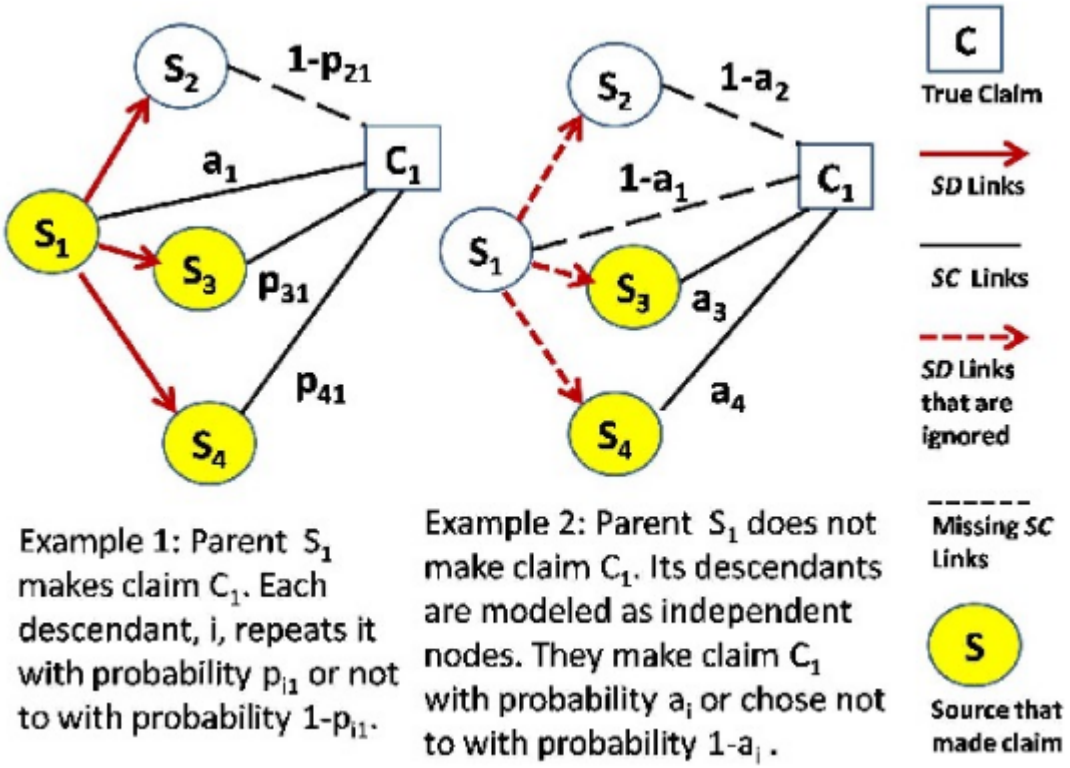
We can assume that the claims are independent, so we can rewrite the probability. $P(SC|SD, \Theta) = \sum Z\ P(SC, z|SD, \Theta)$ where Z is a vector where $zi = 1\ if\ Ci$ is true. Z is a Boolean vector that gives us information about the truthiness of each claim, obviously this vector is not known in advance. If we are able to obtain Z and Θ we reach our objective because we have the reliability of the sources given by Θ and we also know which claims are true and which claims are false.

How can we consider the SD graph? How can we put the information about the relationships among sources into the classical maximum likelihood estimation problem?
THe idea is that we consider the SD graph, where each node is a source and edges are relationships (there are at least 4 ways to build this) and having this we can consider subsections of this graph with a parent and the children. Each subgraph represents a dependency between resources. What does it mean that sources depend on each other? It simply depends on the fact that the information, or observations provided by children are not original.
Another point is that even though relationships exist between parents and children, we cannot assume that a child will always retweet the father's content. It might happen, it might not.
So, considering the social dissemination graph, if the parent does not make the claim, then the children act as if they are independent sources. If the parent makes the claim, then each child repeats it with a given repeat probability.
This can be done by merging graphs SC and SD and dividing them into subsets, SCj, each one describing which sources espoused the Cj and which did not.
Then, we compute the repeat ratio:

$$p_{ik} = \frac{\text{number of times Si and Sk make same claim}}{\text{number of claims Sk makes}}$$

where i is the child and k is the father.

Here we have an example:



We embed the two graphs to see who posted what and who is independent.
In the first example the parent makes the claim, and then each child has a repeat probability p. Otherwise, the children act independently.

In the next image is shown the iterative algorithm used to compute the vector Theta.

Algorithm 1 Expectation Maximization Algorithm

1: Initialize θ with random values between 0 and 1
2: Estimate the dependent ratio (i.e., p_{ig}) from source dissemination graph SD based on Equation (7)
3: **while** $\theta^{(n)}$ does not converge **do**
4: **for** $j = 1 : N$ **do**
5: compute $Z(n, j)$ based on Equation (11)
6: **end for**
7: $\theta^{(n+1)} = \theta^{(n)}$
8: **for** $i = 1 : M$ **do**
9: compute $a_1^{(n+1)}, ..., a_m^{(n+1)}, b_1^{(n+1)}, ..., b_m^{(n+1)}, d^{(n+1)}$ based on Equation (12)
10: update $a_1^{(n)}, ..., a_m^{(n)}, b_1^{(n)}, ..., b_m^{(n)}, d^{(n)}$ with $a_1^{(n+1)}, ..., a_m^{(n+1)}, b_1^{(n+1)}, ..., b_m^{(n+1)}, d^{(n+1)}$ in $\theta^{(n+1)}$
11: **end for**
12: $n = n + 1$
13: **end while**
14: Let Z_j^c = converged value of $Z(n, j)$
15: Let a_i^c = converged value of a_i^n; b_i^c = converged value of b_i^n; d^c = converged value of $d^{(n)}$
16: **for** $j = 1 : N$ **do**
17: **if** $Z_j^c \geq 0.5$ **then**
18: C_j^* is true
19: **else**
20: C_j^* is false
21: **end if**
22: **end for**
23: Return the claim classification results.

- The input is the source claim graph *SC* from social sensing data and the source dissemination graph *SD* estimated from social network
- The output is the maximum likelihood estimation of source reliability and claim correctness

At line 17 we have to set a threshold to determine if a claim is true or not. This approach allows us to infer the credibility of an observation using a huge number of observations, and to be more optimistic or pessimistic. This approach demonstrates to work better than the voting approach.

When we evaluate a method, we must use the baseline approach. The natural baseline is the voting method - we have a claim, we count the number of votes attributed to that claim, and we decide with a threshold if that claim is true or not. How can we estimate data credibility - so, which claims are true, and which ones are false? Voting: the number of times that the same tweet is collected from the human network, the larger the repetition, the more credibility is attributed to the content.
As we said before, the weakness of voting is that it counts both original tweets and retweets, so it cannot distinguish between original information and dependent information. We could try by only including original tweets to have a better estimation of the credibility.

Another baseline is the standard Expectation Maximization, that is: substantially the same algorithms that the authors used, but without considering the SD graph, so we are not able to distinguish between reliable sources and independent sources.
So, we collect the claims and gain data credibility by maximizing the likelihood of tweets collected from the human network, assuming that all sources constitute independent observers.
We can distinguish between regular-EM and EM with admission control. The first one assumes that all sources constitute independent observers, while the second one removes dependent sources using some heuristic approaches from social networks.

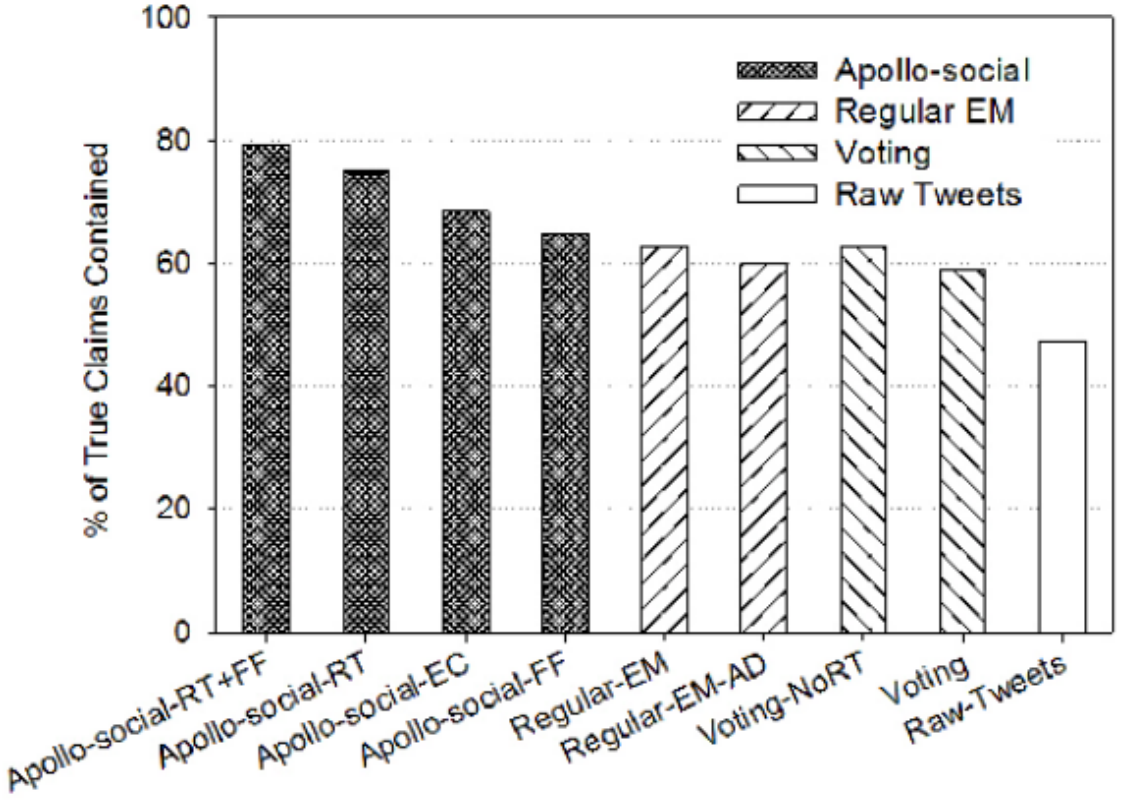
The general methodology, in practical terms, uses these steps.

1. Choose the event type of interest
2. Select keywords and, possibly, geographic location
3. Crawl Twitter using the available API
4. Log collected tweets
5. Apply tweet clustering to determine claims
6. Build SC and SD graphs
7. Apply MLE filtering to eliminate the false claim, so in the output of the process we only have claims that are likely to be true.

What can we do to validate the methods? If we do not have the ground truth, we must build one! That is, consider the collected dataset and manually label each claim true or false. Of course, we must limit the number of claims to consider here.
In the paper, the output of filtering was manually graded to determine match with ground truth. Due to man-power limitations, manually grading only the 150 top ranked claims.

In this manual grading, we can decide whether something is true or not by, for example, looking at sources external to the experiment, like mainstream media sources.
What do we do with unconfirmed claims? We can decide to drop claims or to infer that it is true or false. If we drop the unconfirmed claims, we are following a pessimistic approach. If your method behaves well in a pessimistic scenario, you can try to take back some dropped claims.

What about the precision of the method that we discussed?



What we see in this image is the percentage of true claims that the system, called Apollo, is able to provide. This graph was computed using the manually labeled ground truth datasets.
Raw tweets give us no way to understand the credibility of the claim.
Voting is a little better, Regular EM is similar to voting in terms of performance (but the voting approach is much faster!!! It takes seconds, whilsts EM takes one hour).
Apollo with the retweeting network and the follower-followee network reaches the 80% of true claims, which is very good also considering that it uses a pessimistic approach - what this means is that it could work even better in a real world scenario.

Some observations and limitations of Apollo.
The scheme tends to reduce the number of introspective (e.g., emotional and opinion) tweets, compared to tweets that presented descriptions of an external world: the reason may be that emotions and slogans tend to be retweeted and hence, tend to be suppressed by the scheme. In contrast, external facts are often observed independently by multiple sources, and hence not suppressed.
Distortions and biases of human sensors are quite persistent in terms of direction and amplitude, unlike white noise distortions. In a sense, humans exaggerate information in predictable ways.

The FF, RT and RT+FF schemes can be improved by building information propagation models that account for:

- topic (e.g., sensational, emotional, and other news might propagate along different models)
- expertise

- relationship reciprocity
- mutual trust
- personal bias (e.g., the claim that «police is beating up innocent demonstrators», versus «demonstrators are attacking the police»)

meme di incoraggiamento per aver finito di leggere gli appunti: