UNIVERSITY OF TARTU
FACULTY OF MATHEMATICS AND COMPUTER SCIENCE
Institute of Computer Science

Kaarel Hanson

Context Sensor Data on Demand for Mobile Users Supported by XMPP

Master Thesis (30 EAP)

Supervisor: Satish Narayana Srirama, PhD
Co-supervisor: Huber Flores, MSc

Author:.................................. ".....” May 2012

Supervisor:.............................. ".....” May 2012

Co-supervisor:........................... ".....” May 2012

Professor:................................ ".....” May 2012

TARTU, 2012

Abstract

Nowadays, technological achievements in user context monitoring techniques enable automating certain computational tasks which are executed by anticipating the user's intention. Smartphones enrich the mobile applications with proactive behavior in usability that allows fitting contextual requirements in real-time. Generally, such behavior is achieved by the mobile device itself through the use of embedded micromechanical artifacts that enable perceiving the environment. Furthermore, mobile applications can benefit from distributed pervasive services located in the environment in order to enhance the mobile experience of the user, like in the case of context-aware games, domotic applications, etc. However, pervasive services for mobile users are constrained to the hardware limitations of the electronic appliances (e.g. CPU power, memory, storage, energy, etc) used for sensing the context. Thus, pervasive services are not able to scale on demand and perform data-intensive processing. To overcome the issues regarding scalability, data integrity and processing power deficiencies and to enrich the smartphone applications with detailed and consistent contextual information, current thesis proposes an optimized implementation of XMPP for transporting sensor data from Arduino microcontroller to the cloud. Arduino provides low-cost hardware, while the cloud offers the reliable and high-availability means for storing and processing sensor data.

Contents

List of Figures

1

Introduction

1.1 Introduction

Nowadays, technological achievements in user context monitoring techniques enable automating certain computational tasks which are executed by anticipating the user's intention. This kind of techniques are based on the processing of sensor information that is collected from multiple sources such as smartphones, environment, etc.

In the case of smartphones, sensor information is gathered by embedded micromechanical artifacts (e.g. accelerometer, gyroscope, etc) and processed locally in real-time (in the phone) for changing some usability aspects in the mobile applications. For instance, the accelerometer sensor can be used for rotating the screen of the device depending on how the user is holding the handset. Another example is the light sensor which enables augmenting and decreasing the brightness of the mobile screen according to the situation of the environment (e.g. indoor, outdoor, etc).

Similarly, environmental sensor information is provisioned as a service (raw data) to mobile users by locating multiple microelectromechanical appliances within the environment. For instance, a thermistor sensor can be located for perceiving the temperature in the context of the user so that a mobile application can use that information for a proactive reaction (e.g. displaying different background screens or triggering a vibration event in the case of mobile pervasive games).

However, more sophisticated sensor mining approaches that enable extracting relational trends, combining data from multiple sources (e.g. motion detection combined with light sensing for determining occupancy) and identifying more complicated patterns (e.g. indoor positioning) require more computational power and storage space. In this context, mobile applications are constrained to the implementation of such techniques due to the limited capabilities of the handset (e.g. battery, etc).

Furthermore, electronic appliances that are used for sensing information from the context (aka smart technologies) and publishing pervasive services for mobile consumption are not scalable (amount of concurrent users) and lack of quality of service. Moreover, such systems are usually costly to implement, and thus they are not feasible in all kinds of scenarios.

On the other hand, cloud computing is arising as a uniform and ubiquitous technology that allows deploying services on the fly without managing the underlying technology, scaling the applications based on demand (multi-tenancy) and benefiting from the pay-as-you-go utility model in order to reduce costs. Mobile technologies are drawing the attention to the cloud due to the demand of the applications, processing power, storage space and energy saving.

Collecting sensor data with the mobile and sending it to the cloud has been simplified significantly as most of the mobile platforms support the implementation of embedded databases (e.g. SQLite) and offloading of data based with REST clients.

However, transportation of sensor data from the environment to the cloud is a complex process that involves working with low-level power devices limited in memory, in storage, etc. Moreover, environmental information is sensitive to drastic changes and, thus an optimized approach that allows monitoring the context in real-time must be considered.

In order to investigate the transportation of sensor data to the cloud from electronic appliances such as those positioned in isolated locations, this thesis studied XMPP protocol as a light and real-time mechanism for data communication. Once in the cloud, the environmental sensor data may be processed (e.g. MapReduce), combined, etc, for mobile consumption using push technologies (e.g. AC2DM, APNS, etc.) or middleware frameworks (such as MCM) that allow establishing asynchronous communication with the handset.

1.1.1 Motivation

Pervasive services for mobile users are constrained to several hardware limitations such as memory, storage, etc. Thus, such kind of services are not able to scale on demand. Moreover, smart technologies are rather expensive. Consequently, their adoption is far from being ubiquitous.

Alternatively, some low-cost technologies, such as Arduino, can be implemented for the consumption of context-aware services. However, they suffer from the same hardware deficiencies.

1.1.2 Contributions

To overcome the issues regarding scalability, data integrity and processing power deficiencies and enrich smartphone applications with detailed and consistent contextual information, current thesis proposes an optimized implementation of XMPP for transporting sensor data from Arduino microcontroller to the cloud. Arduino provides the low-cost hardware, while the cloud offers a reliable and dependable means for storing and processing sensor data.

In order to prove our concept, we have implemented XMPP on an Arduino Mega ADK and extended the OpenFire server (running on Amazon) with a dashboard that relies on a database (MySQL) for storing the data sent by Arduino. Once the data is gathered by the dashboard, it can be routed to a specific cloud service (e.g. S3 or EC2).

The prototype was extensibly analyzed, in terms of energy consumption (Arduino) and concurrent users (OpenFire dashboard). According to the results, XMPP enables to transport data to the cloud with considerable ease.

1.1.3 Outline

Chapter 2: discusses the state of the art addressed by this thesis. First, the chapter introduces the protocol in question (XMPP), cloud and mobile computing terms are presented. Later, some cloud terminology is introduced. Finally a brief overview of Arduino framework is given.

Chapter 3: explains the problems regarding provisioning of pervasive services to mobile users and the transportation issues of sending environmental data to the cloud from electronic appliances.

Chapter 4: addresses the realization details of the prototype. The chapter first introduces the architecture of the prototype. Later, describes the fundamentals of XMPP together with examples of communication messages. Further, gives details about the implementation of the complete framework, the XMPP client on Arduino and the OpenFire dashboard. The chapter is concluded by discussing some issues occurred during the development.

Chapter 5: introduces couple of real-life scenarios and their analysis (energy consumption and server scalability).

Chapter 6: provides the conclusion of the results achieved of the thesis together with future research directions.

Chapter 7: discusses related work of the field.

2

State of the Art

The continuous improvements in real-time communication have provided us the means for ubiquitous presence in our daily lives. One of these achievements is instant messaging (IM). Basically, IM consists of several clients (buddies) that exchange information between them using a central server (e.g. OpenFire, etc.) that manages/orchestrate the communication. The most well-known examples are MSN, AOL , Yahoo Messenger and Jabber.

In this chapter, we present a complete description of Jabber and pointed out some opportunities that emerge when extending the protocol to transport sensor data to the cloud. Finally, we focus on describing Arduino microprocessor as open source electronic platform and its integration with the Android mobile platform.

2.1 Jabber

Jabber or XMPP(aka eXtensible Messaging and Presence Protocol), is a near-real-time communication protocol that relies on Extensible Markup Language (XML) and enables the exchange of structured data between any number of network entities (1). XMPP protocol was developed by the Jabber open-source community in 1999, initially as an instant messaging protocol. Since then the technology has been extended from network management systems to online gaming among others.

The protocol enables the exchange of small pieces of structured data (called "XML stanzas") between two or more entities over the network. XML streams and XML stanzas make possible the rapid, asynchronous and synchronous exchange of data between XMPP entities. XMPP employs the use of Transport Layer Security (TLS) and Simple Authentication and Security Layer (SASL) to ensure the security and authenticity of the communication channel. There are three core stanza types - presence, message and iq - discussed in further sections.

XMPP uses decentralized client-server model, where each user connects to the server that controls its own domain. There is no central authoritative server as in the case of MSN, AOL or ICQ messengers. Moreover, there are many open-source implementations of XMPP servers that can be deployed in a private domain. For example, by encapsulating XMPP to transmit over another protocol (for example HTTP) for fitting specific requirements in the rules of firewall, one can deploy a server for internal communication within an organization and extend its communication with another server entities that are spread over a large geographical area.

XMPP can be envisioned as a push technology to deal with offline operations (asynchronous operation). In this kind of operation, XMPP delivers the message to the server, so that once it has reached there, the input stream is delivered to the corresponding buddy depending on his/her status (online, offline, etc.). If the status is offline, the message is located in a queue and will be delivered once the user comes online.

Since XMPP has been around for a while, additional requirements and functionality has been added over the time. These have not been added to the core XMPP but are documented as XMPP extension protocols (XEP).

2.2 Mobile Computing

Improvements in mobile device technology, hardware (embedded sensors, memory, power consumption, touchscreen, better ergonomic design, etc.), software (more numerous and more sophisticated applications due to the release of iPhone (2) and Android (3) platforms) and transmission technology (higher data transmission rates achieved with 3G and 4G technologies) have contributed towards having higher mobile penetration and better services provided to the customers. Also, those improvements have enabled the mobiles to become the source of information and understand the user in multiple ways (interaction, movement, location etc.) (4).

2.2.1 Smartphones

A smartphone is a device that extends the capabilities of mobile phone by adapting a higher application layer that enables managing any artifact attached to the mobile resources. Modern smartphones usually have high-end touchscreens and high-speed data access via Wi-Fi or mobile broadband (e.g. 3G/4G).

Smartphones are equipped with embedded cameras and micromechanical artifacts such as accelerometer sensor, proximity sensor, gyroscope sensor, compass and global positioning system among others. The number and type of sensors may vary depending on the manufacturer and the model of the device itself. For example, the gyroscope is a fairly new addition that is only available for a few handsets such as Samsung Galaxy S2 and Apple iPhone 4S.

2.2.1.1 Accelerometer

The accelerometer is an artifact that measures the acceleration of a device which it has been embedded in. Usually a triaxial accelerometer is incorporated in smartphones, and thus acceleration can be tracked along the x, y and z axis. Each axis measure is related to the movement towards a single direction (forward/backward, right/left and up/down). For example, in the case of a person walking in a museum tour, forward/backward is related to speeding up and slowing down, up/down involves going upstairs or downstairs in the museum, and left/right is recorded when the person is making turns while walking (5).

2.2.1.2 Magnetic Field

Magnetic field sensor (aka hall sensor) is used for sensing magnetic disruption in the environment. Depending on how disruptive is the measurement, this information can be used for approximating orientation and direction of the device.

2.2.1.3 Gyroscope

The gyroscope sensor is an actuator based on the principles of angular momentum conservation that is used for establishing position, navigation and orientation of the device, among others. It consists of three axis or freedom degrees (spinning, perpendicular and tilting) mounted in a rotor which consists of two concentrically pivoted rings (inner and outer). The gyroscope is used within the mobile for enhancing techniques such as gesture recognition and face detection. Furthermore, the combination of accelerometer and gyroscope sensor data increases motion accuracy, and thus techniques such as video stabilization are implemented on the mobile (6).

2.2.2 Mobile platforms

Mobile platforms are the base for every mobile device. They provide SDK, the tools and the operating system that make it possible to develop applications for that platform. Usually they have their own distribution model which is supplied by the platform developer. For example Apple iOS has AppStore, Google Android has Play Store and Windows Phone has Windows Marketplace. Mobile applications are developed for a diversity of mobile platforms including Android, iOS (iPhone), Symbian (Nokia), Windows Mobile (Microsoft), Blackberry (Sony Ericsson), etc. Recently, Symbian has been replaced by Windows Mobile since the popularity has decreased in the mobile market.

Since the introduction of iOs (2007) and Android (2008) as mobile platforms for smartphones, huge popularity has been growing around these two operating systems as basis for the development of mobile applications. Apple released iOs as a platform for their own devices (iPhone, iPod Touch). In contrast, Android was released as open source by the Open Handset Alliance. Currently, Android is supported by several vendors such as Samsung, Sony Ericsson, HTC, Toshiba, LG among others.

2.2.2.1 Android

Android is a mobile platform released by the Open Handset Alliance that consist of a software stack composed by an operating system, middleware and key ap-

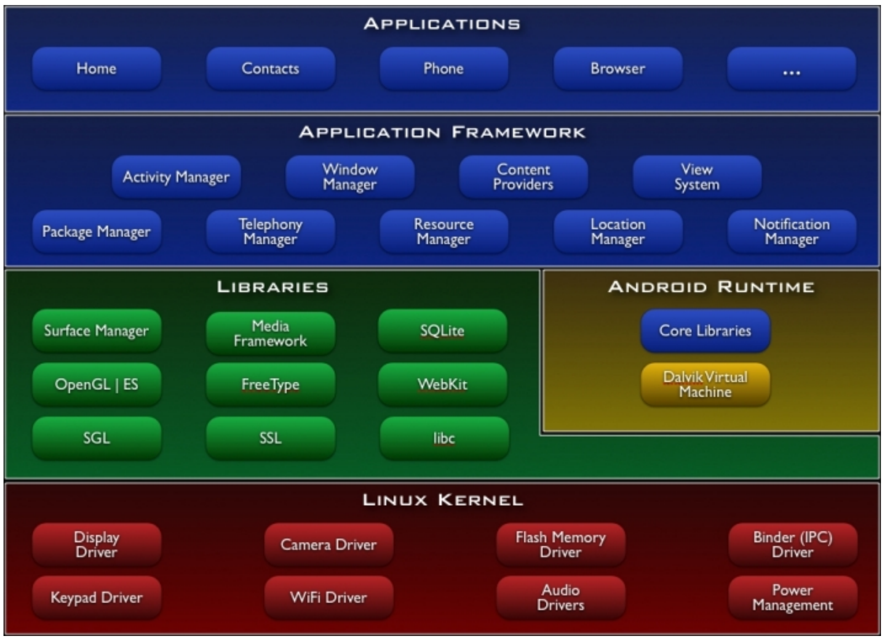


Figure 2.1: Android Architecture

plications (3). The development of the software is tied to the use of the Dalvik Virtual Machine (Android Runtime) that enables using Java as programming language. Most of the libraries that are compatible with JDK can be deployed on an Android device. However, some of them may present issues concerning compatibility with the compiler, and thus are unable to execute. For example, typica API, a Java library for accessing Amazon Web Services, presents such integration problems.

Android architecture consists of multiple layers that rely on a Linux kernel as shown in Figure 2.1. The application layer includes a set of default applications in the operating system such as calendar, contacts, etc. Those applications can be synchronized with the cloud using Google Sync. The application framework consist of predetermined services for managing hardware resources (sensors, screen, etc), software (alarms, background services) and integrating with external resources (location information systems, AC2DM notification service, etc).

In its native libraries, Android incorporates applications that provide excellent real time performance. An example of such application is the light version of SQLite database.

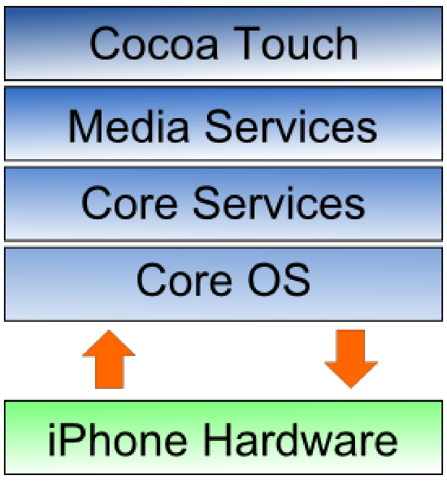


Figure 2.2: iOS Architecture

Although, Android applications are mainly distributed in the Android Market, applications can be distributed freely over the Internet once they are packed in APK (Android Application Package) files. However, if the application is not purchased in the Android Market, there is the risk of acquiring Android malware.

2.2.2.2 iOS

iOS is a mobile platform created by Apple, Inc and is deployed on its mobile devices (iPhone, iPod Touch). Since it is a proprietary technology, most of its core functionality altogether with the hardware is not accessible to the developer. iOS consists of multiple software layers, each allowing application development.

iOS architecture is depicted in Figure 2.2. The complexity of each layer is related to the lines of code needed to achieve the objective in the mobile application. In general, the higher the level, less effort required for building the application.

Cocoa Touch Layer is the highest layer of the iOs platform, written in Objective-C language. The layer provides services such as Push Notification Service, Game Kit Frameworks, among others. Media Layer provides capabilities for reproducing audio and video together with graphics capabilities for animations. The Core OS Layer lies at the bottom of the iOS stack and, as such, sits directly on top of the device hardware. The layer provides a variety of services including low level networking, access to external accessories and the usual fundamental operating system services such as memory management, file system handling and threads.

The distribution of applications is exclusively arranged through the iPhone AppStore. Developer who desires to publish applications must first submit them for inspection to the iOS Dev Center. Once the review is completed and the application fits the mandatory requirements set by Apple, the application is published to the AppStore.

2.3 Cloud Computing

Recently, there has been growing interest in adapting the cloud computing paradigm for delivering services that require high demand. Services provided through the Internet are moving to the cloud due the scalability benefits in its infrastructure, the ease of the deployment of services without managing the underlying technology (software and hardware) and the pay-as-you-go model that enables reducing costs. Business companies are also migrating to the cloud to leverage the potential of their internal infrastructure (using the computing power when needed). Cloud services can be provided by public cloud vendors (Amazon AWS, Google App Engine, Microsoft Azure, etc.) or own private cloud implementations (Eucalyptus). Cloud domain is strongly dominated by proprietary solutions (public clouds). Hence, there exist various cloud architectures that may use different styles (SaaS, PaaS and IaaS) for delivering the cloud supplies. Such architectures are accessible through particular implementations, API set, etc, provided by one specific vendor.

2.3.1 Cloud Services

Cloud services are provided on demand at different levels. Figure 2.3 shows the layers of cloud services, in terms of level of abstraction. The provisioning of services can be at the Infrastructural level (IaaS), Platform level (PaaS) or the Software level (SaaS). In IaaS, commodity computers, distributed across the Internet, are used to perform parallel processing, distributed storage, indexing and data mining. IaaS provides complete control over the operating system and the clients can fully benefit from the computing resources like processing power and storage. One IaaS level provider is Amazon EC2 (7).

Virtualization is the key technology behind realization of these services. PaaS mainly offers hosting environments for other applications. Clients can deploy domain specific applications on these platforms, one of which is Google App Engine (8). These applications are in turn provided to the users as SaaS.

SaaS is generally accessible from web browsers, e.g. Facebook. Web 2.0 is the main technology behind the realization of SaaS. However, the abstraction between the layers is not concrete and several of the examples can be argued for other layers.

While there are several public clouds on the market, Google Apps (Google Mail, Docs, Sites, Calendar, etc), Google App Engine (provides elastic platform for Java and Python applications with some limitations) and Amazon EC2 are probably the most popular and widely used. Elastic Java Virtual Machine on Google App Engine allows developers to focus on developing applications rather than bother about maintenance and system setup. Such sandboxing, however, places some restrictions on the allowed functionality. Amazon EC2 on the other hand allows full control over the virtual machine, starting from the operating system. It is possible to choose a suitable operating system and platform (32 or 64 bit) from many Amazon Machine Images (AMI). In addition, there are several possible virtual machines that differ in CPU power, memory and disk space. This structure of service enables choosing the right resource for any particular task. In the case of EC2, price of the service depends on the machine figures, its uptime, and used bandwidth, both into and out of the cloud.

2.3.2 Cloud Providers

2.3.2.1 Amazon

Amazon (7) AWS (Amazon Web Services) is a public cloud computing provider that offers a variety of infrastructure and platform services (e.g. Hadoop) over the Internet. Amazon AWS relies on the top of the cloud computing infrastructure for delivering services that can be accessed using REST (REpresentational State Transfer) and SOAP (Simple Object Access Protocol). Within the bundle

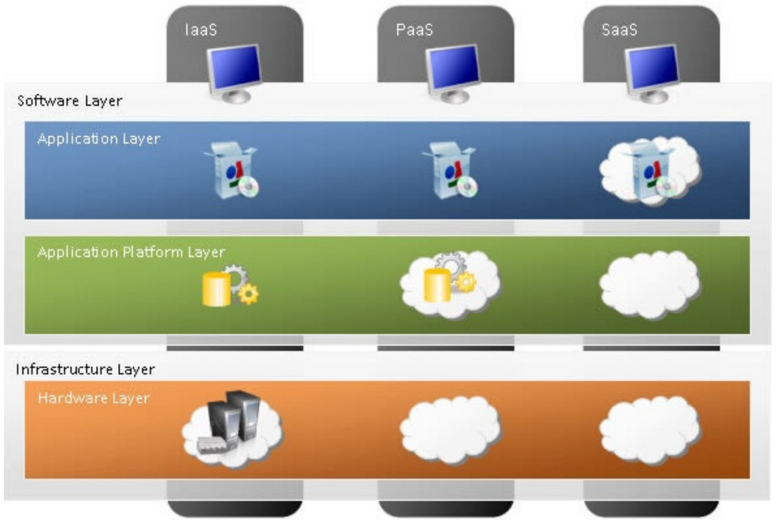


Figure 2.3: Levels of Abstraction in Cloud Computing

of services provided in the Amazon stack, EC2 (Elastic Compute Cloud) and S3 (Simple Storage Service) can be mentioned as these are the most popular and well-known services. Other services have been developed around these basic services such as EBS (Amazon Elastic Block Store), AWS Management Console, etc. Moreover, one of the latest services provided by Amazon is CloudWatch for monitoring the applications that are running in the cloud.

Amazon services are paid according to the user's consumption of resources (number of requests, amount of bandwidth, etc). However, in February (2011), Amazon released a free tier account for the developers in order to foster the creation of applications based on their cloud infrastructure.

EC2 EC2 (Elastic Compute Cloud) is the central part of Amazon Web Services. EC2 provides the means to rent instances of virtual machines for specific purpose. For example, it can be used for implementing parallel computing algorithms with distributed frameworks such as Hadoop among others.

Instances are launched on demand, which in turn enable creating highly scalable web applications. Furthermore, EC2 gives the users control to choose the geographical location of instances in order to optimize latency and therefore allow high levels of redundancy.

S3 Amazon S3 (Simple Storage Service) is a scalable, high-speed, low-cost

Web-based service offering means for data storage and backup, and web and image hosting. The service can be connected to over Web service interfaces, e.g. SOAP, REST. Furthermore, BitTorrent protocol is supported to provide high-scale distribution at lower cost.

S3 allows uploading, storage and downloading of practically any file or object up to five terabytes (5 TB) in size. Amazon imposes no limit on the number of items that a subscriber can store and claims that subscribers have access to the same storage infrastructure Amazon uses to run its own Web applications.

2.3.2.2 Azure

Windows Azure is Microsoft's platform for cloud computing. Windows Azure was created with the purpose to simplify IT management and minimize upfront and ongoing expenses. By design, Azure facilitates managing of scalable Web applications over the Internet. Microsoft data centres take care of and maintain the hosting and management environment.

The platform can be used to create, distribute and upgrade Web applications without the necessity to maintain the resources in use. Web services and applications can be created and debugged with ease and low personnel expense. What is more, new capabilities can be added "on the fly" to existing packaged applications.

2.3.2.3 Google App Engine

The Application Engine contains all the services provided by Google. It uses a SaaS approach for delivering services over the Internet. Among the most well-known services are Google Analytics, Google docs, Picasa, etc. Google App Engine also supports data storage (Google for developers).

Android mobile platform is tied to the solutions provided by Google, and thus most of the services released over the Internet have been extended to provide a mobile version. As an example, Android applications, such as Calendar, E-mail, Contacts, can be easily synchronized with Google services if the user has a Google Account.

2.3.2.4 Eucalyptus

Eucalyptus is a platform for the implementation of private cloud solutions. It offers an enterprise solution for business in general and a version for the open-source community. Eucalyptus architecture is based on Amazon, and thus share similar composition for provisioning of services. Walrus provides the support for storage (similar to S3) and Eucalyptus provides the computational service (like EC2). However, applications that are developed for Eucalyptus are not highly compatible with Amazon. Therefore, applications must be modified is the necessity to migrate from one architecture to another.

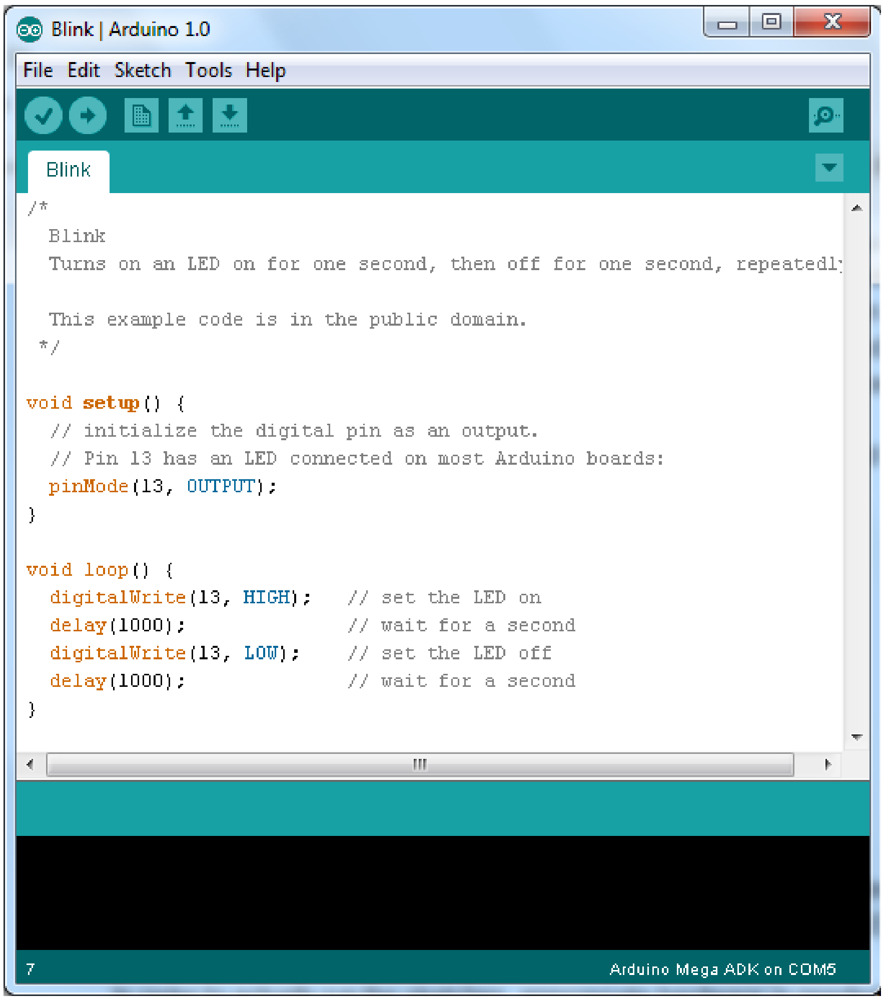
2.3.3 Cloud and XMPP

There exist public clouds and private clouds. Public clouds are free for use for anyone who needs to use the available resources offered by a service provider, similar to ISP-s offering connectivity to the Internet. Private clouds are deployed by organizations for their private purposes, e.g. Eucalyptus. Then there are Intercloud Exchanges where the clouds can interoperate. To tie the clouds together a central and structured system is needed. Intercloud Root is such an instance, providing all kinds of root services, including Naming Authority, Trust Authority, Directory Services, etc. In order to establish secure and reliable communication between the instances, a suitable protocol is required. There are popular standards and protocols, like HTTP, for such a job. Unfortunately, the synchronous communication that Hyper Text Transfer Protocol (HTTP) offers is unsuitable for time-consuming operations, like computationally demanding database lookups. Server timeouts are common obstacles for such operations. In contrast, XMPP based services are capable of asynchronous communication and are ideal for lightweight service scenarios. In (9) XMPP is employed as an Intercloud communication and service request protocol. As a viable control plane presence and dialogue protocol, it is perfect for this purpose. XMPP allows cloud instances to dialogue with each other and find one or more clouds that are ready and willing to exchange any kind of information. In addition, the protocol supports many-to-many messaging across service provider domains and can be used to carry messages of different types.

Bioinformatics widely utilize clouds and Web services technologies like Simple Object Access Protocol (SOAP) and REpresentational State Transfer (REST). These technologies have some severe drawbacks, including lack of service discovery and inability to send status notifications (10). A problem is that SOAP services, which are common in the field, are typically using HTTP as communication channel. Complex communication is common in bioinformatics, however, HTTP was designed to accommodate query and web pages retrieval. Also, the SOAP specification does not by itself provide means of service discovery. To overcome the drawbacks, IO Data, which is an extension to XMPP, has been taken into use. The difference between HTTP and XMPP is their wrapping - HTTP is unstructured while XMPP is formatted as XML, allowing integration with the abundance of XML languages used in bio- and cheminformatics. The extension enables services to publish their own input and output data types, asynchronous invocation, and allows existing XMPP infrastructure to discover services. Besides the aforementioned, XMPP supports determining their status and availability. Although, XMPP cloud services offer numerous advantages, there are some limitations when transferring large data for two reasons. First, all data transmitted with XMPP must be wrapped in XML which requires data conversions that might be less efficient when compared to a plain binary stream. Second, each XMPP message has to wait for its own turn to be transmitted over the XMPP stream. In case a large message is being sent, the following messages are retained until the transmission of the large message is completed.

2.4 Arduino

Arduino (11) is an open-source electronics prototyping platform based on flexible, easy-to-use hardware and software. It's intended for anyone interested in creating interactive pervasive applications. Arduino senses the surrounding environment via a variety of sensors and can control various gadgets, such as lights, relays and servos. The microcontroller on the board is programmed using the Arduino programming language (based on Wiring) and the Arduino development environment (based on Processing). The program that can be written and uploaded to the microcontroller with Arduino IDE is called a sketch. The sketch has two



```
/*
  Blink
  Turns on an LED on for one second, then off for one second, repeatedl

  This example code is in the public domain.
 */

void setup() {
  // initialize the digital pin as an output.
  // Pin 13 has an LED connected on most Arduino boards:
  pinMode(13, OUTPUT);
}

void loop() {
  digitalWrite(13, HIGH);   // set the LED on
  delay(1000);              // wait for a second
  digitalWrite(13, LOW);    // set the LED off
  delay(1000);              // wait for a second
}
```

Figure 2.4: Arduino IDE with Sample Sketch

mandatory functions - *setup* and *loop*. When Arduino is started or restarted, the first function called is *setup*, which usually contains settings initializations (e.g. Serial port). If *setup* has finished, loop function is called over and over again without any delay until Arduino is restarted or has crashed. In the latter case the program is restarted. Crashes can occur if main memory use is exceeded or a pointer error occurs. Arduino sketches are written in C/C++ and support most of standard C/C++ libraries.

Figure 2.4 shown above depicts Arduino IDE 1.0 with a simple Arduino sketch that manipulates an LED light. In the *setup* method, pin 13 is initialized as output. This is the pin that an LED light is connected to. In the *loop* method, the output power for pin 13 is set to maximum and then back to minimum. One second delay is set between each of the commands. The *loop* method runs infinitely. In order to actually run the sketches, appropriate hardware is needed. Arduino offers several different microcontroller boards, some of which are Uno, Duemilanove and Mega ADK.

2.4.1 Arduino Mega ADK

Arduino Mega ADK board, based on the ATmega2560, has the best technical characteristics of all the Arduino boards. It has the largest SRAM (8 KB), flash for storing code (256 KB of which 8 KB is used for bootloader) and EEPROM (4 KB) memory. It is equipped with a USB host interface to directly connect with Android based devices. The board has 54 digital input/output pins, 16 analog inputs, 4 UARTs (hardware serial ports), a power jack for external power source and a reset button. The Mega ADK board is based on an earlier Mega 2560 board. Similarly to the Mega 2560, it features an ATmega8U2 programmed as a USB-to-serial converter. This artifact is used for connecting with the computer to upload sketches.

The Arduino ADK can be powered via the USB connection or with an external power supply. The power source is selected automatically. External (non-USB) power source can be either an AC-to-DC adapter or battery which can be connected by a 2.1mm center-positive plug. Arduino Mega ADK microcontroller can be seen in Figure 2.5.

2.4.2 Android and Arduino

The USB accessory mode allows connecting host hardware using USB, but the limitation is that the used hardware must be specifically designed for Android-powered devices. However, if the accessories adhere to the Android accessory protocol outlined in the Android Accessory Development Kit documentation, we can still use the Android-powered device to interact with USB hardware, although, the device cannot act as a USB host. Android turns normal USB relationship upside-down and the accessory added to the Android powered device acts as host and the Android device itself is the USB Device. The USB accessory

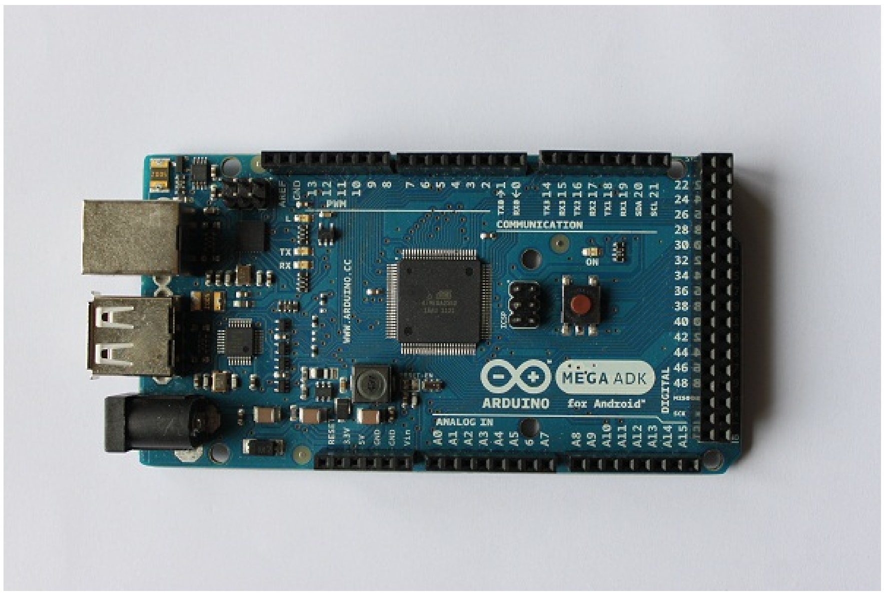


Figure 2.5: Arduino Mega ADK

APIs were introduced in Android version 3.1, however, it is available in Android 2.3.4 using the Google APIs add-on library.

It is possible to establish a connection between Arduino microcontroller and Android powered device when using Android Open Accessory Development Kit or other USB Host Shield compatible with Arduino. Android Accessory library can be used to implement the protocol. The most suitable microcontroller is Arduino Mega ADK, which is specially developed for working side by side with Android powered devices.

2.4.3 TinkerKit

TinkerKit (Figure 2.6) is a set of electronic sensors and actuators mounted on boards that can be hooked up to the Arduino via a Sensor Shield. It enormously simplifies electronic prototyping, because the unnecessary and time consuming soldering and sensor building out of breadboard, wiring and resistors is skipped. The kit has been created for education and design, allowing setting up interactive and smart environments quickly. The shield used to test the implementation is called Mega Sensor Shield, which is specifically created for Arduino Mega boards,

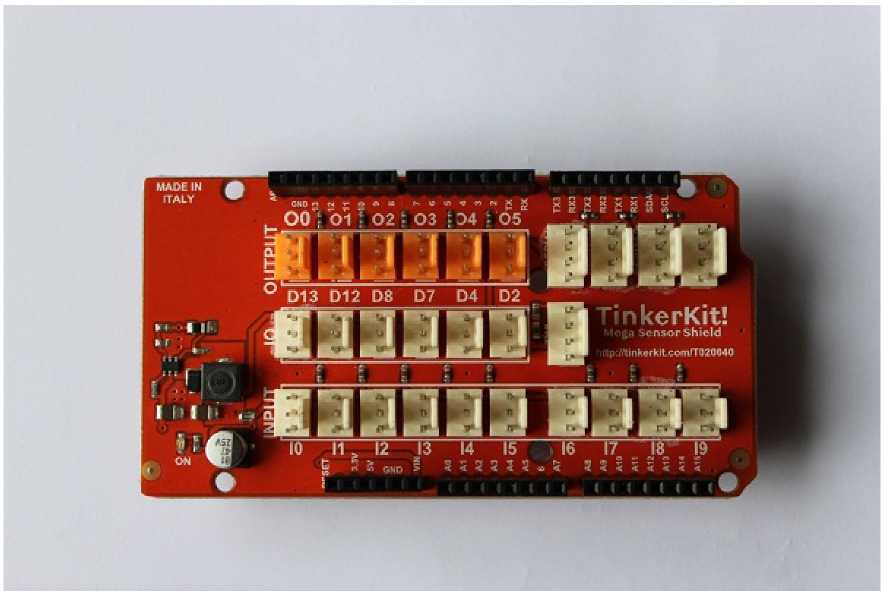


Figure 2.6: TinkerKit Mega Sensor Shield

regarding the physical dimensions.

2.4.3.1 Sensors

TinkerKit offers various kinds of different sensors that enable reading information from the environment as well as influence the environment via lights, motors and control other electronic devices via relays or mosfets.

Thermistor Thermistor sensor has been designed to measure temperature of the room or any location for that matter. Electromechanically, this sensor decreases the resistance of the circuit as the temperature rises, therefore opening up the circuit. The raw output reading is between 0 and 1023, but TinkerKit library provides functions for converting the readings to Celsius and Fahrenheit degrees.

Light Dependant Resistor Light dependant resistor is used for measuring light intensity in the environment. When no light is falling on the sensor, the resistance is maximum and the circuit is closed. If the light becomes more intense, the resistance starts decreasing and the circuit is opened up. Maximum voltage this sensor outputs is 5V and expected reading is from 0 to 1023.

2.4.4 Shields and Modules

Several shields and modules have been developed for specific purpose that can be attached to the microcontroller. These components make hardware assembly a lot simpler and can be directly attached on top of the microcontroller without the need of building one out of breadboard, wiring and resistors. Some popular shields are Wireless, Ethernet, Motor and Proto Shield. These are manufactured by Arduino but addition to them there are many other makers, like Sparkfun, DFRobot, and shields, like GPS, MP3 and LCD shield.

2.4.4.1 Ethernet Shield

The Ethernet Shield is used to connect to the internet via CAT5E cable with RJ45 jack. It supports both 10 and 100 MB connection speeds, operating voltage needed is 5V and connects with Arduino on SPI port. The shield is based on the Wiznet W5100 ethernet chip (12). Both TCP and UDP are supported by the network stack on the chip and is capable of maintaining up to four simultaneous socket connections. The connection between an Arduino board and the shield is enabled by long wire-wrap headers which extend through the shield. This keeps the pin layout intact and allows another shield to be stacked on top. For convenience, there is an official Ethernet Library for managing internet connection, using Arduino as a client or a server. In addition, the shield offers micro-SD card connectivity, which comes in handy when serving files over the network is required. Arduino Ethernet Shield can be seen in Figure 2.7.

2.4.4.2 Wireless Shield

The Wireless Shield (Figure 2.8) enables communicating using a wireless module. Like the Ethernet shield, it has an onboard SD-card connection capability for serving data on the card over the network.

The shield has an on-board switch labelled Serial Select. It determines how the Wireless Shield's serial communication connects to the serial communication between the microcontroller and USB-to-serial chip on the Arduino board.

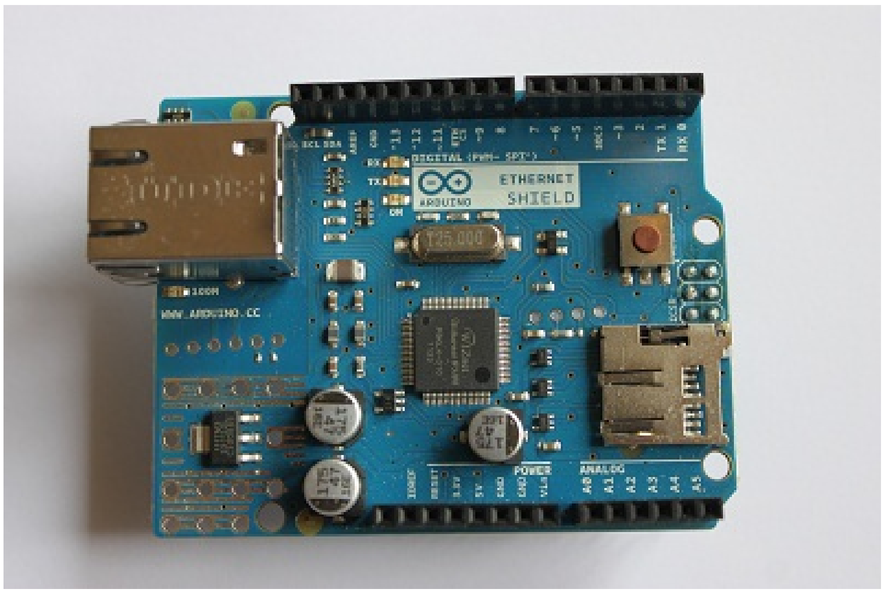


Figure 2.7: Arduino Ethernet Shield

The switch has two settings - Micro and USB. In Micro mode, the wireless module communicates with the microcontroller. Data sent from the microcontroller will be transmitted to the computer via USB as well as being sent wirelessly by the wireless module. In this mode the microcontroller cannot be programmed via USB. In USB mode, the microcontroller on the board is bypassed and the module can communicate directly with the computer.

When configuring the module, we need to remember to upload an empty sketch to the microcontroller before. Such a sketch consist only of empty *setup()* and *loop()* methods. This is the case since the configuration is done over Serial communication port (COM), but also the microcontroller communicates with the wireless module over the same port. When an active sketch is running on the microcontroller, it is going to interfere.

2.4.4.3 WiFly Wireless Module

The RN-XV WiFly radio module (13) is a standalone embedded Wi-Fi access device pre-loaded with manufacturer firmware simplifying integration and development time. It is based on Roving Networks RN-171 robust Wi-Fi module and

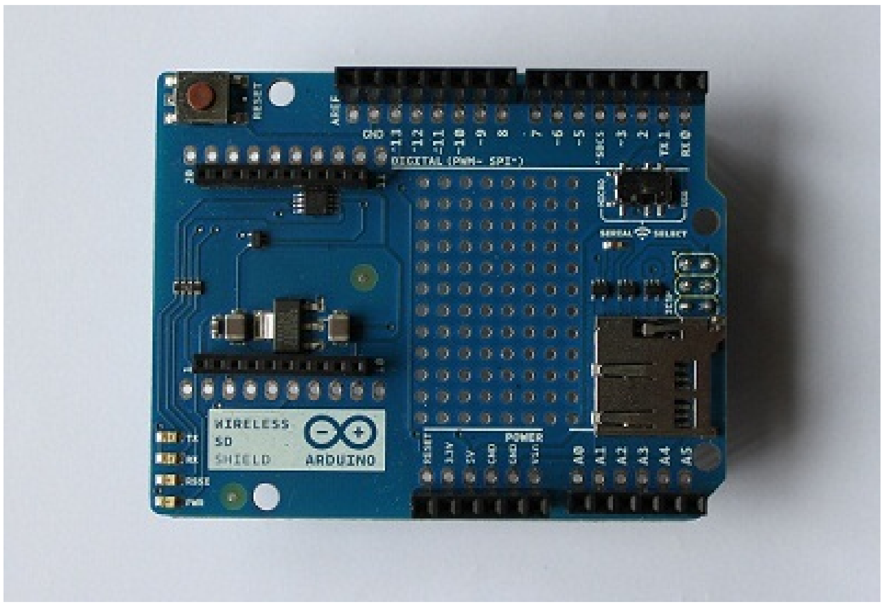


Figure 2.8: Arduino Wireless Shield

is equipped with 32 bit processor, TCP/IP stack, real-time clock, crypto accelerator, 8 Mbit flash memory and 128 KB RAM. Only initial configuration is required to access network and start sending and receiving serial data over UART. The module requires quite low power and has several transmit power levels from 0 to +12dBm. It can be configured over Wi-Fi or UART using ASCII commands. Several Wi-Fi authentication algorithms are supported, including WEP, WPA-PSK (TKIP), WPA2-PSK. The nature of the module enables it to use in wide range of applications including industrial metering, room temperature sensors, pump configuration and control, telemetry and robotics. The module is shown in Figure 2.9.

2.4.5 Arduino and XMPP

At the time of writing there are no public XMPP implementations for Arduino. There is some code published at GitHub (https://github.com/adamvr/XMPPArduino) but unfortunately it does not work with the latest Arduino IDE 1.0. Quite several public implementations have been created that execute on the computer, request sensor data from the microcontroller over Serial and then send it to an XMPP

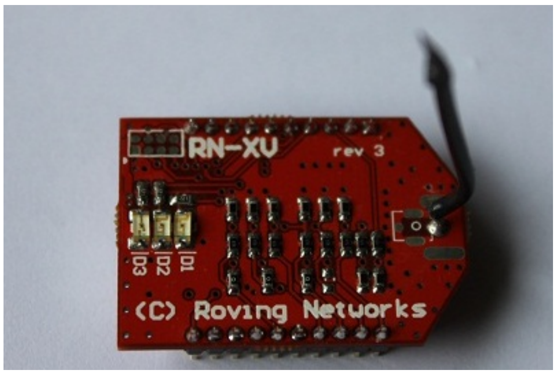


Figure 2.9: Roving Networks WiFly RN-XV Wireless Module

server like OpenFire. This kind of approach does not deplete the whole potential that Arduino hardware and framework offer. Otherwise we could just attach sensors directly to the computer, read data and send it to the web or even process and present it on the same computer. The whole idea behind Arduino is that it is an independent entity that can be physically placed to unimaginable locations and does not need to depend on the resources provided by the computer.

Although there is not anything similar developed for Arduino, there is an XMPP implementation created for Contiki operating system (14). Contiki is an open-source and portable operating system, designed mainly for memory-constrained network systems and embedded systems on microcontrollers. This approach relies on info/query requests received from XMPP clients over the Internet in order to obtain the sensor data.

2.5 Summary

In this chapter, there were several prominent technologies outlined, which can be combined to provide better user experience and enrich mobile applications. XMPP protocol was described as a promising communication protocol as the basis of real-time sensor data transmission. A brief overview of smartphones, smartphone embedded sensors and mobile platforms, including Android and iOS, was given. Both private and public clouds were discussed together with specific vendors. Furthermore, an overview of XMPP as a cloud service communication protocol was presented on the example of the two existing cloud infrastructure implementations. In the last subsection, Arduino and related hardware, including communication shields and sensors, were reviewed.

3

Problem Statement

Even though, the integration between mobile devices and electronic appliances is feasible for creating pervasive mobile applications, the hardware limitations of the appliances (e.g. memory, storage, etc) restrict the solutions to scale on demand. Moreover, smart technologies are rather costly and far to become ubiquitous.

In this chapter we present the drawbacks that emerge when providing environmental information to smartphones from electronic platforms such as Arduino. Finally, we introduce some possible solutions to tackle those issues.

3.1 Environmental Sensor Data for Mobile Users

Nowadays, smartphones enrich the mobile applications with proactive behavior in usability that allow fitting contextual requirements. Generally, such behavior is achieved by the mobile device itself through the use of embedded micromechanical artifacts that enable environment perception. Furthermore, mobile applications can benefit from distributed sensors located in multiple areas in order to enhance the mobile interactive experience, like in the case of pervasive games, domotic applications, etc.

However, pervasive services for mobile users are constrained to the amount of users that can be handled (due hardware limitations). Thus, such kind of services are not able to scale on demand. Furthermore, Smart technologies are rather expensive. Consequently, their adoption is far from being ubiquitous.

Alternatively, some low-cost technologies, such as Arduino, can be implemented for the consumption of context-aware services. However, Arduino is not reliable and scalable enough to deal with a huge number of simultaneous users. For instance, the Wiznet W5100 chip on the Ethernet Shield supports only up to four concurrent socket connections and the WiFly module is limited to one, which is below expectations for a server. This is feasible for personal use but for wider audience is out of question.

Data storage problems are also going to be addressed. Although, Arduino can save sensor data to an SD card, it does not have the means to do complex data analysis of the environment with historical data, if needed. Furthermore, the data on the SD-card could get corrupted or destroyed altogether, if something happens to the Arduino. For example if the Arduino is set up in an outdoor environment, the hardware can be damaged by the climate. Therefore, to ensure data integrity, it is lucrative to save the data on the cloud.

To address most of these problems, this thesis proposes the implementation of a low-cost system based on Arduino and Cloud technologies that is able to serve a huge amount of concurrent users. The system relies on XMPP in order to establish the communication between Arduino and cloud. XMPP is based on jabber technologies (open-source) and it was preferred in order to avoid the effect of polling (caused by protocols such as HTTP) and thus, increase the battery life. The system takes care of sending sensor information to the cloud storage. Once at the cloud, anything can be done with the data - use sensor data to predict, recognize or detect multiple environmental patterns (e.g. create statistics about weather in certain locations, etc). The possibilities are practically endless.

Moreover, XMPP server can be integrated with other XMPP servers to publish the data to other domains with little to no effort, thus, providing an easy and flexible approach to maintain, port and combine sensor information with other solutions.

3.2 Summary

To overcome scalability, data integrity and processing power deficiency problems and enrich smartphone applications with detailed and consistent contextual information, current thesis proposes a solution to gather data from the sensors and deliver the information from Arduino microcontroller to the cloud using XMPP protocol. Arduino provides a low-cost hardware, while the cloud offers a reliable and dependable means for storing and processing sensor data, and XMPP fulfills the requirements for successful data transportation.

4

From Arduino to the Cloud using XMPP

There are several issues that were discussed in the previous chapter, regarding the utilization of pervasive services within the mobile applications. Current solutions that target mobile consumption, are not scalable on demand (e.g. hardware limitations) and at the same time are costly. In this chapter, the thesis attempts to tackle such issues by proposing a low-cost approach based on Arduino and cloud computing.

In the first part of this chapter, the overall architecture of the solution is described along with some fundamentals of XMPP, in order to understand, how the protocol is adapted to microprocessor for sending sensor information. Later, the chapter provides an overview of the Arduino and server-side clients. Finally, some details about problems/facts that occurred during the development are discussed.

4.1 Architecture and Implementation Details

The implementation of XMPP protocol is written in C++. This is the case because Arduino platform is written in that language. Moreover, this implementation is written for Arduino IDE 1.0, which is the latest version at the time of writing. Some features have changed, new have been added since the last version.

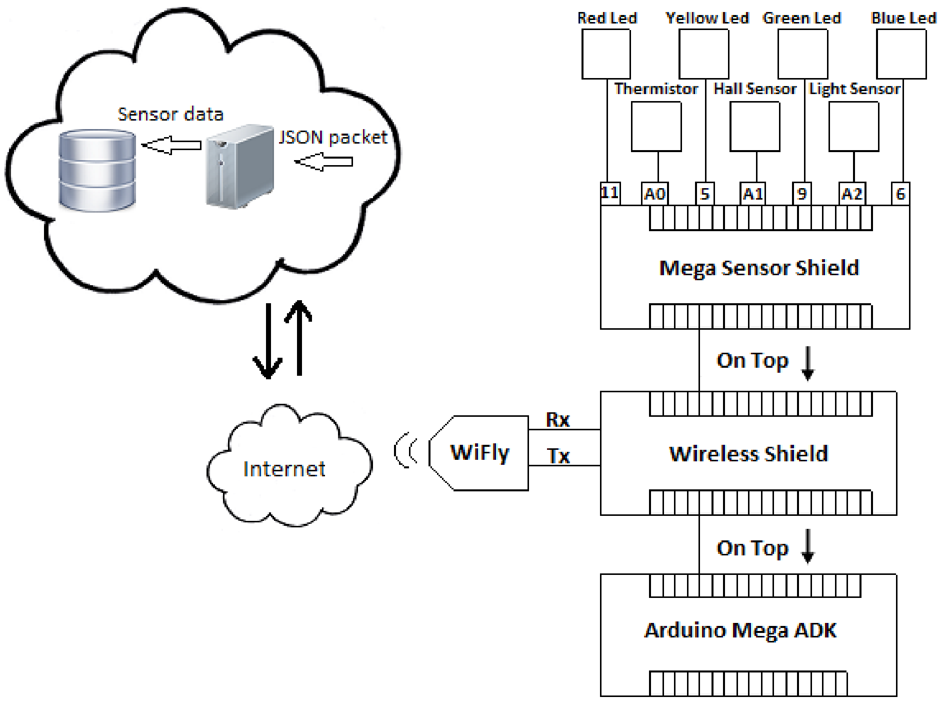


Figure 4.1: Wireless Architecture

The implementation comprises base64 encoder (taken from an XMPP implementation that has been written for earlier Arduino IDE version), sensor protocol precisely created for sensor data transfer, utility methods and XMPP protocol that handle negotiating connection with an XMPP server, stanza creation and message handling. There are two hardware architectures taken into account during the implementation of XMPP protocol. The first one uses Wi-Fi for transmitting data to the server while the other uses Ethernet. Abstract overviews of wireless architecture can be seen in Figure 4.1.

Arduino microcontroller is the bottommost component of the physical hardware setting. Wireless Shield is mounted on top of the Arduino using long wire-wrap headers which extend through the shield. Wireless module is situated on the Wireless Shield and communicates with it over UART. Finally, Mega Sensor Shield is mounted on top of the wireless shield. There are 7 components attached to the sensor shield with cables - four LED lights and three sensors. LED lights are used as indicator lights to show the current state of the sensor. Red indicates that the program has started but network and TCP connection has not

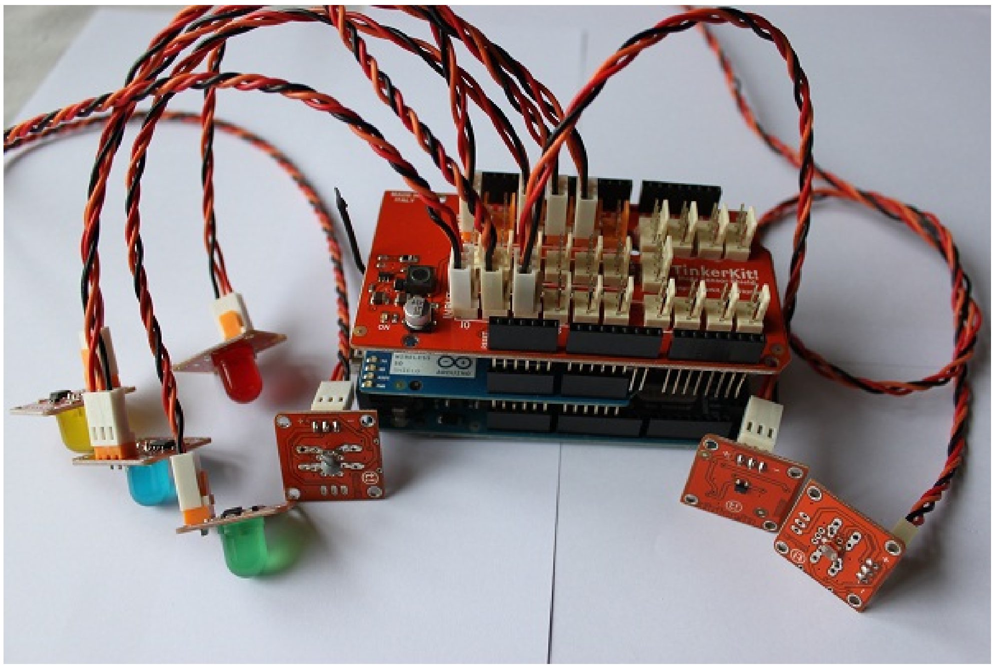


Figure 4.2: Assembled Composition with Wireless Shield

been established yet. When yellow LED turns on, TCP connection with XMPP server has been created. The green light notes that the stream negotiation has been successfully finished and the entity is online. Every time data is sent to the server blue LED flickers. All of the lights are connected to digital pins. Thermistor sensor measures the temperature of the surrounding environment, hall sensor is used for detecting magnetic field and light sensor is a variable resistor, that decreases its resistance when light falls on the sensor. The sensors are connected to analog pins. Physical setting of the wireless system can be seen on Figure 4.2.

The overall architecture of the prototype with Ethernet Shield can be seen in Figure 4.3. The only difference with these two architectures is that the Wireless Shield is swapped with the Ethernet Shield and everything else stays the same. Physical setting can be seen in Figure 4.4.

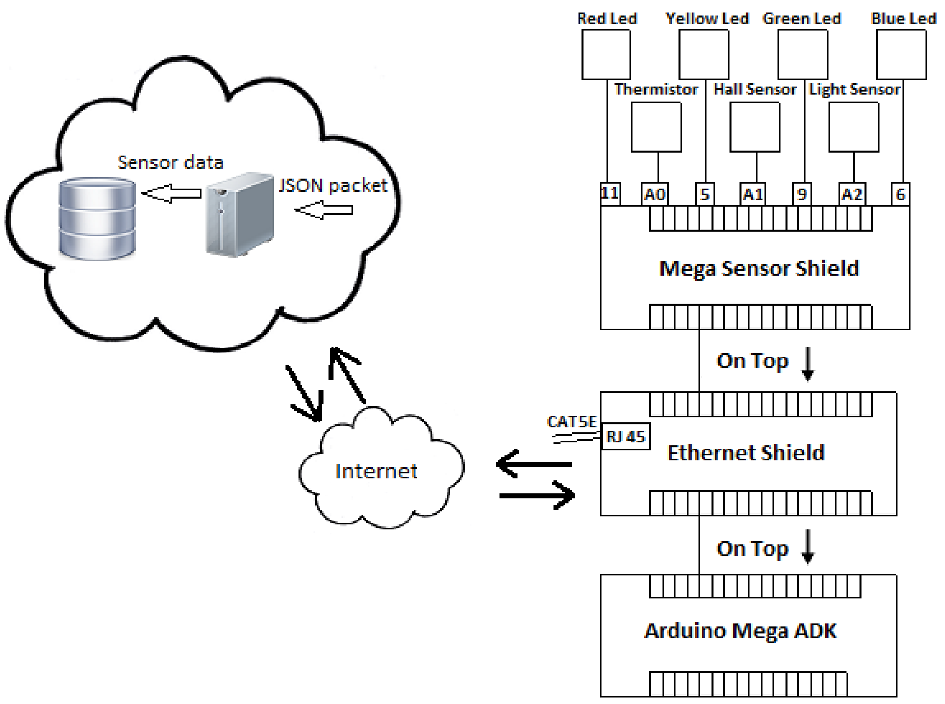


Figure 4.3: Ethernet Architecture

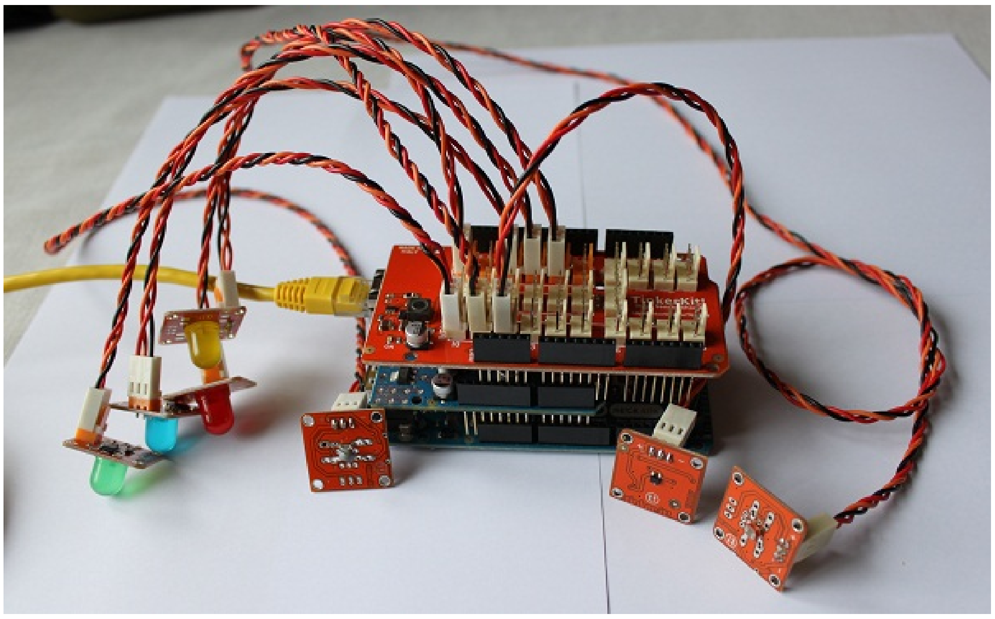


Figure 4.4: Assembled Composition with Ethernet Shield

4.2 Fundamentals

To understand the details of the implementation, one must get acquainted with the fundamentals and essence of XMPP protocol. XMPP functionally is consuming and demanding for low power and minimal resource hardware, in terms of battery and memory, respectively. Therefore only the essential core requirements have been implemented.

XMPP utilizes input/output XML streams during all the communication process. Basically, an XML stream is a container for exchanging XML stanzas between entities. The stream is started by sending a stream header tag (<*stream*>) with appropriate attributes and namespace declarations. During an open stream, the amount of XML elements and stanzas exchanged between entities is not bound. XML stream can be thought of as a large, if not infinite, XML document. The receiving entity must negotiate a stream, called the response stream, in the opposite direction if an entity wants to exchange stanzas with another entity.

An XML stanza is the base of the XMPP protocol. A stanza is a first-level XML element whose element name can be message, presence or iq and whose qualifying namespace is *'jabber:client'* or *'jabber:server'* depending on the entity.

In essence, the communication between client and server can be thought of as two open XML documents. One is acting as an envelope for sending stanzas to the server, the other one acts as an envelope for the XML stanzas received during a session.

4.2.1 Stanzas

After a client and a server have completed stream negotiation, either party can send XML stanzas. There are three different types of stanzas: <*message/*>, <*presence/*> and <*iq/*> (short for info query). The first stanza type is quite trivial, used for pushing information to another entity.

The second stanza is used for sending presence notifications. For example, when a user logs into server, the client sends a presence stanza to the server indicating that the user has changed status. The server sends this status notification to every entity that is online and is in the user's buddy list.

Info query is a request-response mechanism, similar to the Hypertext Transfer Protocol. IQ enable an entity to make a request to and receive a response from another entity. For example, iq is used to request roster (buddy list) after a client has logged into the server.

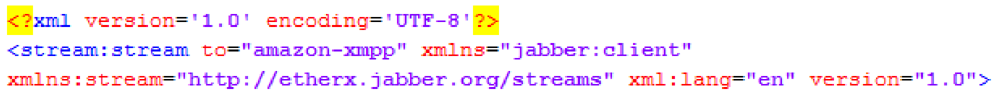
In addition, five common attributes apply to these stanza types. These attributes are *'to'*, *'from'*, *'id'*, *'type'* and *'xml:lang'*. The *'to'* attribute specifies the recipient of the stanza. In contrast, *'from'* attribute specifies who is sending the stanza. *'id'* attribute is used to track any response or error stanza that it might receive related to the generated stanza from an entity. The purpose or context of the message, presence, or IQ stanza is specified by *'type'* attribute. *'xml:lang'* attribute should be added to the stanza if it contains XML character data that is intended to be presented to a human user. The value of the *'xml:lang'* attribute specifies the default language of any such human-readable XML character data.

4.2.2 Address Definitions

XMPP uses globally unique addresses in order to route and deliver messages over the network. All XMPP entities are addressable over the network using unique identifiers. The server identifier is called a domainpart, similar to an e-mail server address, whereas account's identifier is called a JID (Jabber ID). There are two types of JIDs 1) bare JID, and 2) full JID. Bare JID consists of localpart and domainpart (e.g. *arduino@amazon-xmpp*, where arduino is the localpart and amazon-xmpp is the domainpart). Typically a localpart uniquely identifies the entity requesting and using network access provided by a server. Full JID consists of localpart, domainpart and resourcepart (e.g. *arduino@amazon-xmpp/sensor*). Typically a resourcepart uniquely identifies a specific connection (e.g. a device or location) or object belonging to the entity associated with an XMPP localpart at a domain.

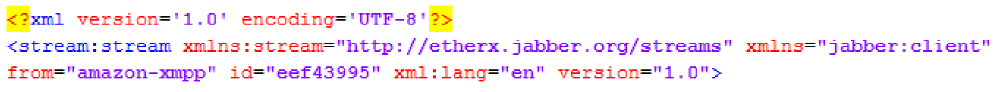
4.3 Session Negotiation

The first thing done in starting the communication is open a TCP connection to an XMPP server. Typically this is initiated by the client, but also can be



```
<?xml version='1.0' encoding='UTF-8'?>
<stream:stream to="amazon-xmpp" xmlns="jabber:client"
xmlns:stream="http://etherx.jabber.org/streams" xml:lang="en" version="1.0">
```

Figure 4.5: Stream Header



```
<?xml version='1.0' encoding='UTF-8'?>
<stream:stream xmlns:stream="http://etherx.jabber.org/streams" xmlns="jabber:client"
from="amazon-xmpp" id="eef43995" xml:lang="en" version="1.0">
```

Figure 4.6: Response Stream Header from Server

created between servers. There are many open and closed-source XMPP server implementations available to choose from. One of the most popular and complete, regarding the features and functionality, is OpenFire. This implementation was also tested with OpenFire XMPP server. Typically, XMPP reserves port 5222.

The XML stream is started when the client sends an unambiguous stream header to the server. This is the root tag of the document, which also contains some necessary attributes and namespace declarations. Example of stream header is shown in Figure 4.5.

The *'to'* attribute has information about the server domain where the client wishes to connect to. The default namespace indicates whether a client or a server is initiating the session, while *stream* namespace declares the XML stream tags. The server responds with similar header, but different attributes, if the initiating header contained correct attribute values (e.g. server domain, namespaces). If the attributes are incorrect, the server closes the stream by sending an unambiguous closing stream tag (*</stream>*), then closes the TCP connection. Example of response stream header is shown in Figure 4.6.

In addition to the stream header, the server sends a first level child element *<features/>* if the stream opening was successful. This tag contains features that the server is offering for the client for the next action. For example, the features contain *<starttls/>* - enables the client to negotiate a TLS encrypted connection -, *<mechanisms/>* - serves the methods or authentication (PLAIN, DIGEST-MD5, etc) -, *<compression/>* - enables transferable data compression. The XMPP protocol specification strongly recommends using TLS to encrypt the communication, but since secure connection creation needs quite a bit of resources (Arduino has an 8 bit processor) and Arduino does not have libraries for

```
<auth xmlns='urn:ietf:params:xml:ns:xmpp-sasl'
mechanism='PLAIN'>AGFyZHVpbm8AYXJkdWlubw==</auth>
```

Figure 4.7: Authentication Stanza with PLAIN Mechanism

```
<success xmlns="urn:ietf:params:xml:ns:xmpp-sasl"/>
```

Figure 4.8: Successful Authentication Message Received from Server

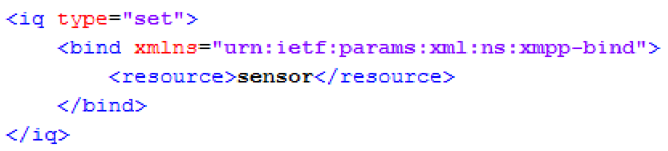
SSL/TLS support, it is unable to use the encryption in this case. Furthermore, SSL/TLS is out of the scope of this thesis. XMPP employs the use of SASL (Simple authentication and security layer) (15) as a mechanism for authenticating the user. The simplest and most insecure mechanism to authenticate is PLAIN (16). In this case the authentication token consist of username and password delimited by NULL character (*'\0'* in C++) in Base64 encoding. Example of an authentication element for username *"arduino"* and password *"arduino"* is shown in Figure 4.7.

Servers turn is to respond whether the authentication was successful or user was not authorized. The stanza received from the server can be seen in Figures 4.8 and 4.9 accordingly.

If the user has been successfully authenticated, the XML stream must be restarted by sending the stream header to the server once again. The server answers in the same way as before but the list of stream features does not contain <*mechanism/*> options anymore since the authentication has been completed already. As one option has been left out, another has been added, namely <*bind/*>. Binding resource is obligatory action and an XMPP server and client must support the implementation. Resource must be bound to the stream so that the server can properly address the client. This means that a resource must be associated with the bare JID (*localpart@domainpart*) of the client so that the address for use over that steam is a full JID of the form *localpart@domainpart/resource*. This ensures that the server can deliver XML stanzas to and receive XML stanzas from the client in relation to entities other than the server itself or the client's

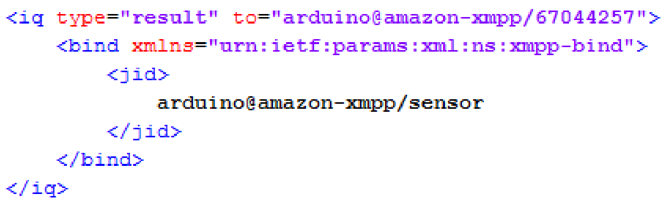
```
<failure xmlns="urn:ietf:params:xml:ns:xmpp-sasl"><not-authorized/></failure>
```

Figure 4.9: Unsuccessful Authentication Message Received from Server



```
<iq type="set">
    <bind xmlns="urn:ietf:params:xml:ns:xmpp-bind">
        <resource>sensor</resource>
    </bind>
</iq>
```

Figure 4.10: Resource Binding IQ Stanza



```
<iq type="result" to="arduino@amazon-xmpp/67044257">
    <bind xmlns="urn:ietf:params:xml:ns:xmpp-bind">
        <jid>
            arduino@amazon-xmpp/sensor
        </jid>
    </bind>
</iq>
```

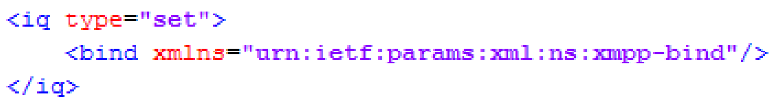
Figure 4.11: Response from Server for Resource Binding

account. After a client has bound a resource to the stream, it is referred to as a connected resource. A server should enable an entity to maintain multiple connected resources simultaneously. An example resource binding iq stanza is shown in Figure 4.10. An example of response from server is in Figure 4.11.

Actually there are two ways to bind a resource 1) explicitly setting the resource name, as show in Figure 4.10, 2) not specifying any resource. In the latter case the server itself assigns a resource to the user. If the user does not fancy it (basically it is a random alphanumeric), it can be changed with the stanza in Figure 4.10. Example command without explicitly setting the resource is shown in Figure 4.12.

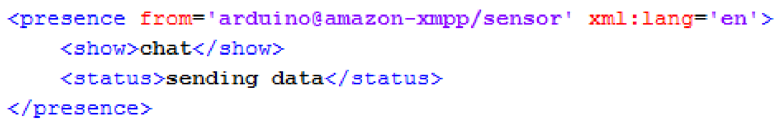
For now the entity has logged in and is able to start communicating with any other entity. Although, one more useful step should be done, send a presence stanza to tell the server that the entity is present and available for chatting. Sample presence stanza is shown in Figure 4.13.

The presence stanza above sets the user's status to chat (meaning available) and status message to "sending data" which is shown in traditional IM clients.



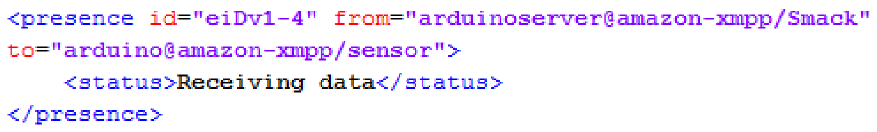
```
<iq type="set">
    <bind xmlns="urn:ietf:params:xml:ns:xmpp-bind"/>
</iq>
```

Figure 4.12: Resource Binding Stanza without Explicit Resource



```
<presence from='arduino@amazon-xmpp/sensor' xml:lang='en'>
    <show>chat</show>
    <status>sending data</status>
</presence>
```

Figure 4.13: Sample Presence Stanza sent when Becoming Online



```
<presence id="eiDv1-4" from="arduinoserver@amazon-xmpp/Smack"
to="arduino@amazon-xmpp/sensor">
    <status>Receiving data</status>
</presence>
```

Figure 4.14: Presence Stanza Received when a Friend Logs in

When the user has sent the presence stanza, the server will respond with presence stanzas for every online buddy that is in the user's buddy list, called roster. Sample stanza of the latter is show in Figure 4.14.

In the example above there is no show element set, since it is optional and therefore is assumed to be online and available. Values that are applicable for show element are

Away the entity or resource is temporarily away

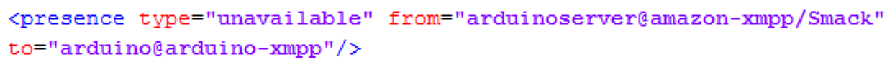
Chat the entity or resource is actively interested in chatting

Dnd the entity or resource is currently busy (Do Not Disturb)

Xa the entity or resource is away for an extended period (eXtended Away)

If an entity or resource logs out from the server, unavailable presence stanza is sent to every buddy, as show in Figure 4.15.

When the presence for the entity or resource has been set, the client is able to exchange unbounded number of XML stanzas with other entities on the network until the stream is closed. The stream closing is rather simple. If the client wishes to log out from the server, it only needs to send the closing tag of the stream (*</stream:stream>*). The closing entity must not immediately close the



```
<presence type="unavailable" from="arduinoserver@amazon-xmpp/Smack"
to="arduino@arduino-xmpp"/>
```

Figure 4.15: Presence Received When a Friend Logs Out

TCP connection but wait for the receiving entity to respond with the same. This indicates that the server has halted any data transmission to the entity and is in state to end the connection. Usually the server closes the connection and the client does not have to do anything. Figure 4.16 depicts the lifecycle of an XMPP session regarding the implementation for Arduino.

4.4 Arduino Client

XMPP Arduino implementation can be used with any network or communication interface that extends Stream class of Arduino base library. Stream class has necessary methods for checking whether there is any data in the receiving byte buffer, reading from the buffer one byte at a time and printing data to the stream.

Since Arduino microcontrollers have little RAM memory (1K to 8K, tested Arduino Mega ADK has 8K) but enough flash memory and to simplify the creation of stanzas, templates of stanzas with format tags are saved in flash memory. The Atmel microprocessors that are used on Arduino microcontrollers have their own library with some useful functions. For example, as mentioned before, it enables saving any type of data in flash memory and the data can be retrieved from there any time needed. This approach is used in the XMPP library.

In Figure 4.17, stream header is saved as char array in program memory, which is the same as flash. Keyword *PROGMEM* indicates that the data is to be saved in program memory. *PROGMEM* is part of the *pgmspace.h* library and it is not automatically included in Arduino sketches but must be included explicitly (*include <avr/pgmspace.h>*). Also it is important to use the datatypes outlined in *pgmspace.h* . Some cryptic bugs are generated by using ordinary datatypes for program memory calls. While *PROGMEM* could be used on a single variable, it is really only worth the fuss if a larger block of data needs to be stored, which is usually easiest in an array.

If an array has been saved in program memory, it must be read back into SRAM when one wishes to use the underlying data. The program space library offers several useful functions for that matter. There are simple string copy functions as well as formatted copy to string. All of the functions are named similarly as the standard C++ functions, only *_P* is added to the end of the

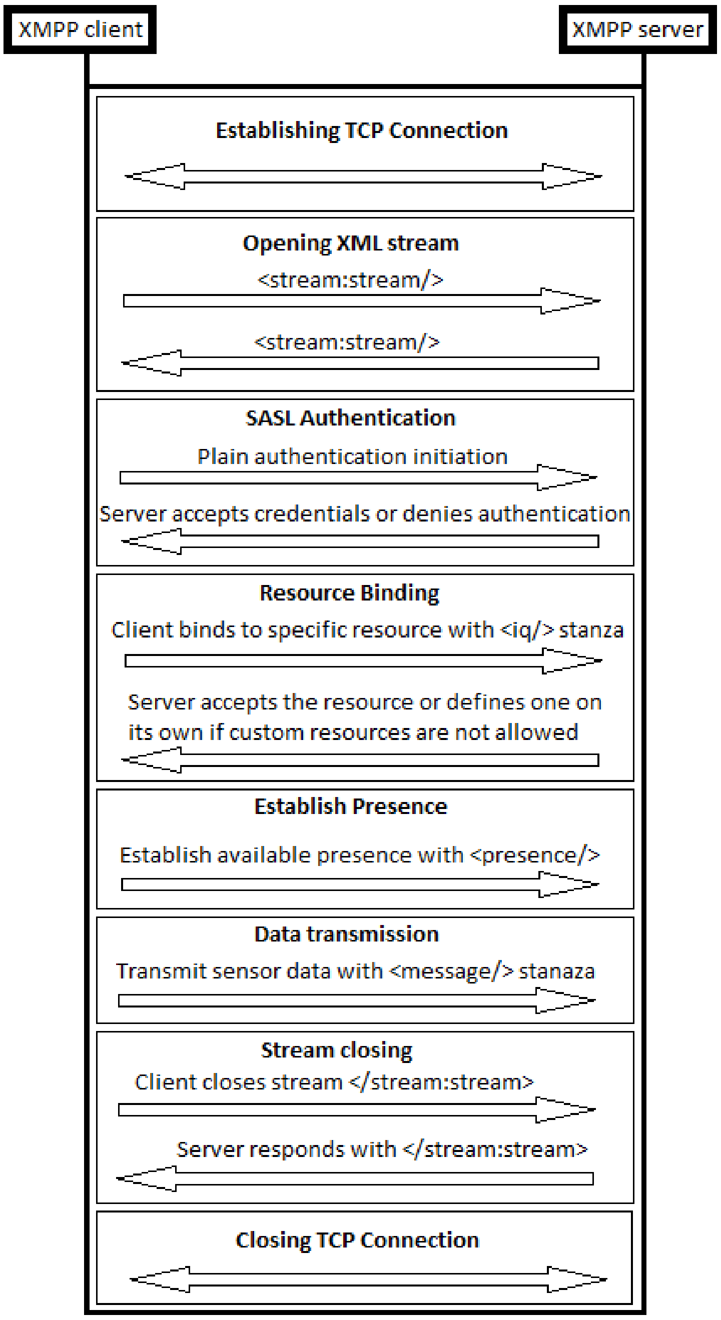

XMPP client
XMPP server
Establishing TCP Connection
Opening XML stream
<stream:stream/>
<stream:stream/>
SASL Authentication
Plain authentication initiation
Server accepts credentials or denies authentication
Resource Binding
Client binds to specific resource with <iq/> stanza
Server accepts the resource or defines one on its own if custom resources are not allowed
Establish Presence
Establish available presence with <presence/>
Data transmission
Transmit sensor data with <message/> stanaza
Stream closing
Client closes stream </stream:stream>
Server responds with </stream:stream>
Closing TCP Connection

Figure 4.16: Lifecycle of XMPP Session

```
prog_char open_stream[] PROGMEM = "<?xml version='1.0' encoding='UTF-8'?>"
    "<stream:stream to='%s' xmlns:stream='http://etherx.jabber.org/streams' "
    "xmlns='jabber:client' xml:lang='en' version='1.0'>";
```

Figure 4.17: Sample Stanza Template Saved in Flash Memory

function name, indicating *PROGMEM* function. To copy data to main memory, the first thing to do is create an array of same standard data type as the type used in program memory. In this case, it is simple char array. The next thing to do is set the size of the buffer the same length as the original array. This can be done with function *strlen_P()*. Finally there are two options 1) do simple string copy, or 2) do formatted string copy. If there are format tags in the array, the simpler option is the second one. Also this is one can cause some trouble, as it was experienced during the development. The destination array's sizes must be set correctly and precisely, otherwise the microcontroller tends to crash or the value of the array after copying does not appear to be the same as the original - several characters missing in the middle, several last characters are omitted. In many cases where this happened, the long way to copy and format the string was used. First, the array was simply copied from program to main memory and then the array was formatted with standard C++ functions (*sprintf()* mainly).

The XMPP library does not handle the connection to the Internet or to an XMPP server. This is the case since one cannot be sure that every communication interface library is built in the same structure or inherit from the same parents. For example, the Ethernet Shield must be started with a method called *begin(byte mac[])* and it requires mac address of the shield as a parameter. On the other hand, the WiFly library has a method called *begin(Stream *serial, Stream* debug)*, which tries to get the connection to the router. The first parameter is the Serial port used for communication between the WiFly module and microcontroller, the second is used for debugging. The library takes into account the fact that Wireless Shield hijacks the main Serial port and therefore cannot be used for both communication and debugging. With WiFly, *begin()* does not establish the connection yet to the router, but initializes the Stream objects and configures the WiFly module. Next, the router is configured (SSID, passphrase) and is joined by calling *join()* method. If the module is previously configured to automatically connect to the router after startup, then there is no reconnection done. Also it is checked whether there is any incoming or outgoing connection. If there is, it is closed because WiFly module supports only one concurrent connection.

XMPP object must be first initialized with the entity's username, password, resource, server domain and the stanza receiving recipient. After the TCP connection has been established the XML stream negotiation is started by calling the *connect()* method of the XMPP library.

Stream negotiation is done in different states. Identifiable states are CLOSED, OPEN, AUTH, AUTH_OPEN, BIND, AVAILABLE and SENDING.

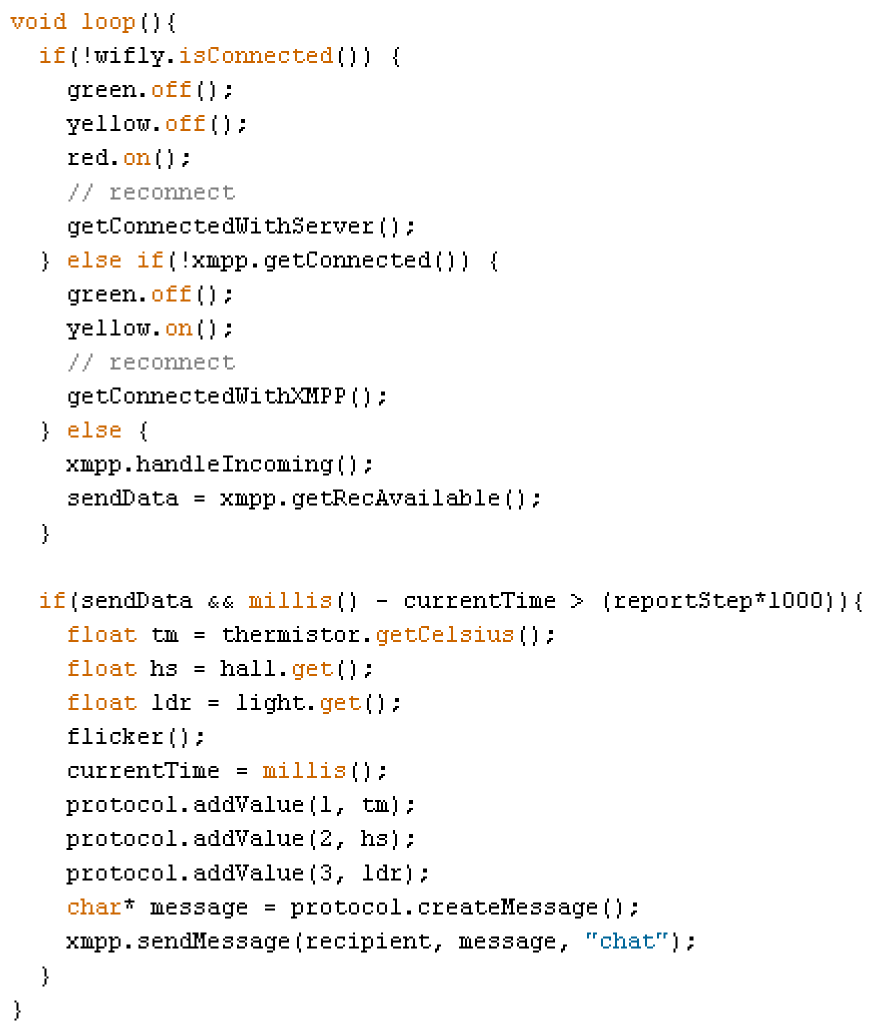
CLOSED state is quite trivial, means that stream negotiation has not been started yet or has been closed already. First state after CLOSED is OPEN. This state is reached when successful response to stream header has been received from the XMPP server.

In general there are many different possibilities what to do in OPEN state but this library currently supports only authentication at this point and only with plain SASL mechanism. None of the other mechanism were implemented because those concern security and improve the confidentiality when transmitting data but they are not relevant to prove this concept. Furthermore, this approach saves precious memory and resources. If authentication stanza gets a success response from the server, current state is set to AUTH.

After authentication the stream is restarted with the stream header as before in CLOSED state. If restart is successful, AUTH_OPEN is obtained. In this state again there are several possible actions. Currently only resource binding is supported. Next, the same resource that was passed as a resource parameter in the XMPP object constructor is bound with the entity. If resource binding is successful, current state is set to BIND. Although, after resource binding there are again several possible actions that are left over from the features list from restarting the stream after authentication, none of them are supported in the prototype.

The last state, that the Arduino can achieve on its own, is AVAILABLE. This state is obtained after presence stanza is sent to the server. At this point Arduino is basically staying idle and the processing is returned to the sketch. To capture any further received packets, *handleIncoming()* must be called from *loop()* method. This method has similar functionality as *connect()*, but this way the control stays in the Arduino sketch. An example of *loop()* with incoming data handling is shown in Figure 4.18.

If presence stanza with recipient credentials is captured, the state is changed to its final possible status, SENDING. Now the microcontroller is reading the sensor



```
void loop(){
  if(!wifly.isConnected()) {
    green.off();
    yellow.off();
    red.on();
    // reconnect
    getConnectedWithServer();
  } else if(!xmpp.getConnected()) {
    green.off();
    yellow.on();
    // reconnect
    getConnectedWithXMPP();
  } else {
    xmpp.handleIncoming();
    sendData = xmpp.getRecAvailable();
  }

  if(sendData && millis() - currentTime > (reportStep*1000)){
    float tm = thermistor.getCelsius();
    float hs = hall.get();
    float ldr = light.get();
    flicker();
    currentTime = millis();
    protocol.addValue(1, tm);
    protocol.addValue(2, hs);
    protocol.addValue(3, ldr);
    char* message = protocol.createMessage();
    xmpp.sendMessage(recipient, message, "chat");
  }
}
```

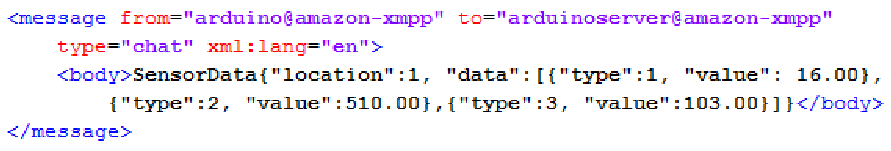
Figure 4.18: Sample loop() Method of Arduino Sketch

data, adding it to the message and transmitting to the server. The transmission delay between the messages is configurable in the sketch. This state is maintained until either the recipient goes offline or the server closes the stream.

In any states if negotiation step fails, the connection is not closed unless the server itself closes the connection. If step fails, processing is returned to Arduino sketch.

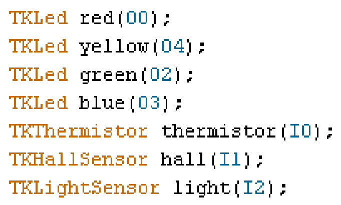
4.4.1 WiFly Libraries

Due to the fact that the RN-XV WiFly module is relatively new (released at the second half of 2012), no official library is present at the time of writing. Therefore the attention is turned to unofficial libraries created by open-source communities. Happily there exist some of these libraries but as expected they have some serious problems and are a bit unreliable. The first library that was discovered is called WiFly-Shield library. At first it showed a lot of potential but after some testing it turned out that the communication between the library and the module was not working properly. For example, the library could not detect whether the module was in command or idle mode and at times tried to configure the module when command mode was not entered yet. In addition, at times joining the access point failed several times before connecting successfully. The module had to be configured for the access point before uploading the program sketch. This was the case because the library was unsuccessful at configuring the module with proper access point credentials due to the communication issues between the library and the module. At the top of these, the library does not properly support hosting Arduino as a server. Another library, called WiFlySerial, that was discovered. This library unfortunately is only usable in cases where the module is manually wired to Arduino and the default Serial port pins (0 and 1) are not used. Since Wireless Shield is used, that approach was discarded and therefore this library is unusable. The last and by far the most reliable library found is called WiFlyHQ. This library did not show the faults described previously. Joining an access point did not pose any problems and the module can be configured from the code. For example, the credentials and security algorithm can be configured in a sketch and anytime Arduino is taken to another access point, only the sketch is needed to



```
<message from="arduino@amazon-xmpp" to="arduinoserver@amazon-xmpp"
    type="chat" xml:lang="en">
    <body>SensorData{"location":1, "data":[{"type":1, "value": 16.00},
        {"type":2, "value":510.00},{"type":3, "value":103.00}]}</body>
</message>
```

Figure 4.19: Message Stanza Complemented with Sensor Data



```
TKLed red(O0);
TKLed yellow(O4);
TKLed green(O2);
TKLed blue(O3);
TKThermistor thermistor(I0);
TKHallSensor hall(I1);
TKLightSensor light(I2);
```

Figure 4.20: Initialization of LEDs, Thermistor, Hall and Light Sensors

be changed. Although it supports hosting Arduino as a server, the connection handling is faulty as is admitted by the developer. The library successfully sends close command to the module but unfortunately the command is discarded. And because WiFly module only supports single connection at a time, the server is in a dead circle, since the old connection cannot be closed and new ones cannot be made.

4.4.2 Sensor Data Transmission

JSON standard is used for sensor data transmission. This approach was taken since it is a lightweight data transmission protocol and provides all the necessary features. Also mapping JSON objects to Java objects on the server-side is very simple due to JSON processor APIs. The main object is named SensorData. It consists of location identification and a list that contains the actual sensor data. Since the data in turn consist of the id of the sensor and the actual reading, another object called Data is required. An example of XMPP message stanza with sensor data is shown in Figure 4.19.

Arduino must be explicitly programmed to use specific sensors. Every sensor has its own object and pin address is required as the only constructor parameter. Sensor initialization is shown in Figure 4.20.

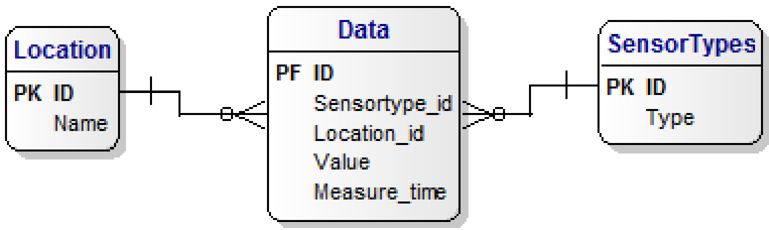


Figure 4.21: Server-side In-memory Database Data Model

4.5 Server-side Client

The server-side client is implemented in Java and uses XMPP API called Smack (?). This API is developed by the same company that developed OpenFire XMPP server. OpenFire server does not need any kind of modifications for this server-side client to work.

Database used for storing sensor data is H2. Due to the proof of concept there was no need to set up full-scale database, but H2 with its in-memory approach is suitable. The database is run in mixed mode, meaning that several connections can be made to it as opposed to embedded mode.

The database contains 3 tables - locations, sensors and data. Locations table contains entries of the locations where the microcontroller can be placed. Therefore, only the id of the location is sent from Arduino, keeping message size smaller. Similarly, sensors table contains rows of different possible sensors that are used for gathering data from the environment. Again, only the id of the sensor is sent over the network to keep the message small. The last table holds the data that is received. Every record contains info about the sensor, location, measured data and the time of arrival. Data model of the database can be seen in Figure 4.21.

For data processing Jackson JSON Processor is used. The processor provides simple JSON to object mapping and only model classes corresponding to the JSON message are needed. The client can be easily modified and enhanced to send the data to S3 bucket or full-scale database and process the data according to requester needs. For example, a weather application could be developed out from this.

4.6 Countered Issues

It seems that the receiving data buffer, in the case of WiFly module, is not filled correctly so the XMPP library must check whether the stanza that has been read is in fact complete and ready for further processing. To check for completeness, it is unwise to compare it with expected char arrays but since it is and XML tag, the easiest way is to count all the signs starting and ending a tag or in other words count the more-than and less-than signs.

If it turns out that the data is indeed complete, it is further processed. If it is not complete, the incoming buffer is checked again whether there are some bytes left. This process is continued until the stanza received is complete.

Another problem that has risen, is that if Arduino has been connected to the server for a while, the connection is dropped, but WiFly does not sense that and keeps sending data, although there is no connection.

Also the data messages seem to be complete when it has arrived to the server. Previously some of the message stanzas were not complete when they arrived. For example couple of characters were missing in the middle of the stanza and because of that the JSON data could not be processed because it was erroneous. This was caused because of the data was divided into small packages of 64 bytes (WiFly's default package size) and some of the characters from the end of a packet went missing since the average stanzas are about 200 bytes. So to transfer 200 bytes, it takes 4 packets by default. To overcome the problem and also maximise the transmission speed, the communication packet size was set to maximum, which is 1460 bytes.

4.6.1 Debugging

Debugging the code when Wireless Shield is connected is not as simple as with Ethernet Shield. The Ethernet Shield does not have the abovementioned Serial Select switch, meaning that the network communication is not done over Serial. Everything that is printed out to the Serial port is not going to be sent over TCP connection to the receiving end. Ethernet Shield uses SPI bus for communicating with microcontroller leaving pins 0 and 1 (RX and TX respectively) untouched.

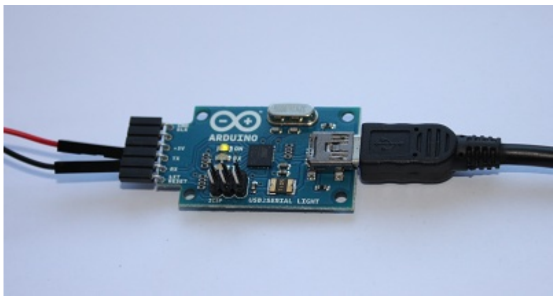


Figure 4.22: USB-to-Serial Adapter

SPI uses digital pins 10 (SS), 50 (MISO), 51 (MISO), 52 (SCK) and 53 (Hardware SS) as communication pins.

This is not the case with Wireless Shield. If anything is printed out to the Serial port, it is directly sent to the receiving party of the underlying TCP connection. This poses a problem since any debugging information sent over TCP could jeopardize the stream negotiation between the XMPP server and client. Fortunately, there are two ways around it.

The first possibility is to wire the pins with Wireless Shield and Arduino manually. This way we can use different pins (default are 0 and 1) as RX and TX pins on the Arduino to avoid hijacking of the Serial by Wireless Shield. This idea seems quite straightforward but it is not that elegant since the shield is built to be mounted directly to the Arduino.

The second alternative is only applicable for Arduino Mega boards. This is the case because these boards are the only ones that have multiple Serial ports - 4 to be exact. In addition to Serial, there are Serial1, Serial2 and Serial3, which have their own RX and TX pins. To use any of the additional ports, a USB-to-Serial adapter is needed. Such an adapter was used in testing the implementation and can be seen in Figure 4.22.

The adapter has a mini USB that is connected to the computer and communicates over COM port, and necessary pins for connecting with the Arduino board. The adapter has ground, 5V, reset, TX and RX pins but we are interested only in the transmit and receive pins because the board is powered by the USB that is connected to itself. While testing the implementation, Serial1 port, which is connected to pins 19 (RX) and 18 (TX), was used.

4.7 Summary

In this chapter was described a prototype solution for a sensor network gathering contextual information from the environment. First of all, an architectural overview of the prototype was given for both Ethernet and Wireless communication interfaces. To understand the details of the prototype, fundamentals (session negotiation, stanzas, address format) of XMPP protocol were discussed.

The solution proposed is based on low-cost Arduino microcontroller hardware and sensors, which is relatively easy to construct and install. To encounter the scalability problem, regarding Arduino, cloud was presented as the central server and data storage facility. In order to tie the components together, XMPP was proposed as an ideal near-real-time communication protocol to deliver the sensor data from microcontroller to server. Moreover, since the prototype relies on instant messaging technique, saving data on the server is done by a second user, server-side client. Due to the proof of concept there was no need to set up full-scale database, but H2 with its in-memory approach was suitable.

What is more, custom sensor encapsulation protocol was created based on JSON format, in order to simplify data mapping on the server side.

Finally, some problems and solutions were discussed that arose during the development of the prototype.

5

Evaluation

5.1 Testing the duration of 9V battery with Wireless and Ethernet interfaces

The tests were carried out with Energizer 9V alkaline batteries (17). In the case of Ethernet the battery lasted for $\approx$ 100.5 minutes, excluding the time needed to connect to the network, server and negotiate the stream. In total the actions sum up to $\approx$ 101 minutes. The microcontroller read information from 3 different sensors - thermistor, hall, light - and for state indication 4 led lights were connected. The data was read and sent to the server every 10 seconds and in total there were 1809 data reads and 603 data transmissions performed by the microcontroller. Therefore we can compute the amount of data that can be delivered with one battery. The stanza sizes and detailed comments about the stanzas can be seen in Table 5.1.

Testing WiFly module with battery was quite troublesome. The main problem arose was that the connection between the Arduino and the server was dropped at some arbitrary time during the test. Although, the indicator lights on the microcontroller denoted that the TCP connection, XML stream were still viable and data was being transmitted, nothing reached the server client. Furthermore, the server had the information that the entity was still online. At first, it was concluded that probably the battery did not have as much power anymore to maintain TCP connection but had enough power to read the data from the sensors

Stanza	**Size**	**Comments**
Header	168 bytes	This stanza is sent twice, at the beginning and again after authentication
Authentication	96 bytes	Username and password used were both "arduino", resulting base64: AG-FyZHVpbm8AYXJkdWlubw==
Resource binding	101 bytes	Resource used was "sensor"
Presence	115 bytes	Included both show and status tags with "chat" and "sending data" accordingly
Message	231 bytes	Included JSON object with data for 3 sensors
Total	139773 bytes	Total includes 603 transmitted message stanzas

Table 5.1: Battery test results with Ethernet connection

and print it to Serial port. This idea arose after testing the wireless module with USB cable. In that scenario the connection was not lost like before. In the last test, the server-side client log was closely followed. Also a packet sniffer utility called *tcpdump* was executed to log any incoming packet on Amazon instance and Tx transmit power level of WiFly module was set to be comparable with the Ethernet Shield (10 on the scale from 0 to 12).

Contrary to the expectations, the battery lasted ≈ 161,5 minutes, excluding connection establishing, although, hardware setup was exactly the same. Altogether with TCP connection creation and session negotiation the result was ≈ 162 minutes. This is more than an hour longer than the Ethernet. The outcome was very surprising, since it was expected that maintaining Wireless communication needed more power compared to Ethernet. During the test ≈ 2907 sensor read operations were called and ≈ 969 messages were delivered from Arduino to the server. The amount of data transmitted can be derived from the number of data transmissions, since the stanza sizes are known. The capacity and comments of the transmitted data can be seen in Table 5.2. Furthermore, the assembled hardware with Wireless Shield powered by the 9V Energizer battery can be seen in Figure 5.1.

Stanza	Size	Comments
Header	168 bytes	This stanza is sent twice, at the beginning and again after authentication
Authentication	96 bytes	Username and password used were both "arduino", resulting base64: "AG-FyZHVpbm8AYXJkdWlubw=="
Resource binding	101 bytes	Resource used was "sensor"
Presence	115 bytes	Included both show and status tags with "chat" and "sending data" accordingly
Message	231 bytes	Included JSON object with data for 3 sensors
Total	224487 bytes	Total includes 969 transmitted message stanzas

Table 5.2: Battery test results with Wireless connection

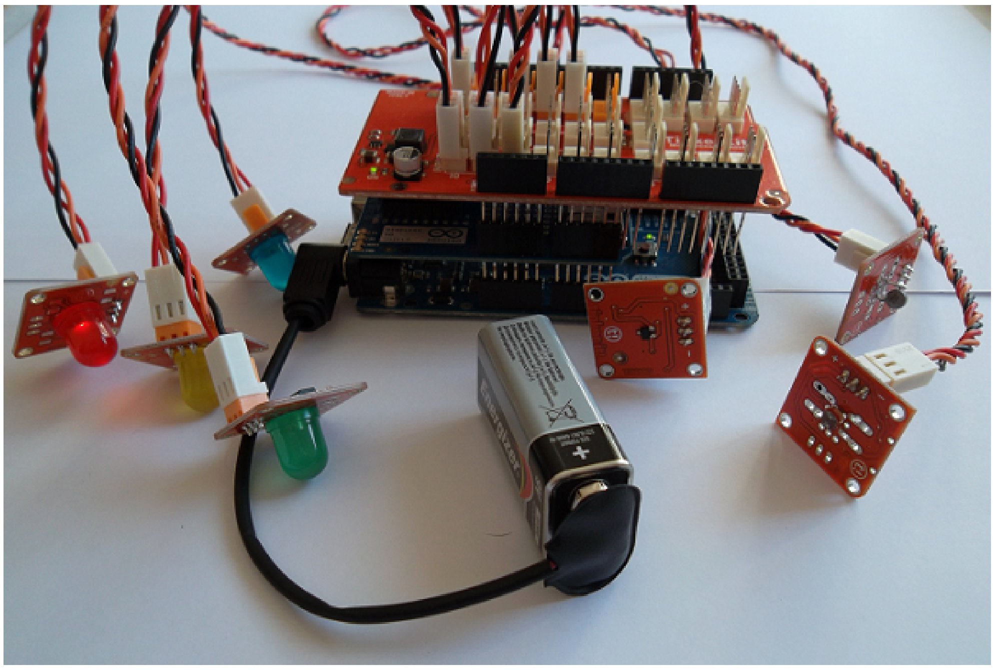


Figure 5.1: Assembled Composition with Wireless Shield and 9V battery

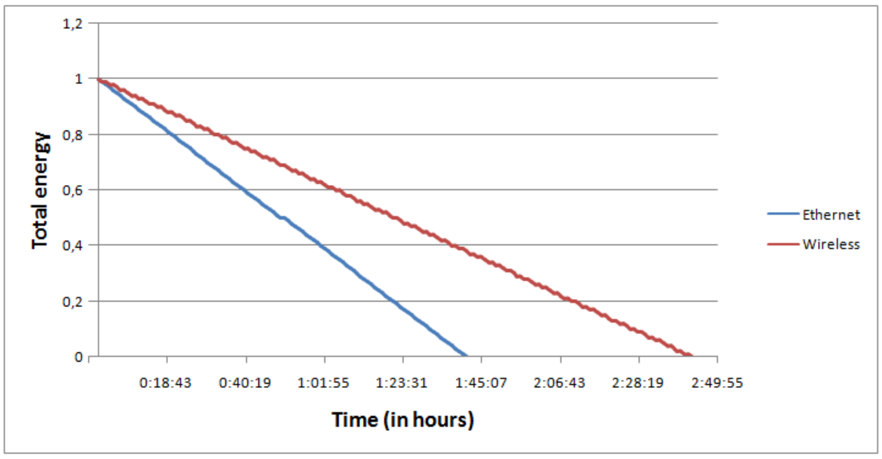


Figure 5.2: Battery Test Results

The collocated results of the two scenarios tested can be seen in Figure 5.2. The energy consumption line is depicted linearly on the graph for illustration, although, in practice the line might be more softer, similar to a curve.

5.2 Load testing OpenFire XMPP server

In order to show the performance offered by OpenFire XMPP server, several load tests were carried out. There were two scenarios considered - replicated dashboard and single dashboard. The dashboard consists of a full-scale MySQL database. In the case of a single dashboard, all of the OpenFire servers share the same transactional space. In contrast, replicated dashboards contain their own database for every individual OpenFire instance (databases are synchronized later). Tsung, an open-source implemented load testing tool, and HAProxy, a high performance TCP/HTTP load balancer, are considered for applying heavy loads to the whole system.

Tsung was deployed on single virtual machine instance in the Amazon EC2 Cloud. The test plan is structured by blocks and consists of three parts, the server/client configuration, in which the machines' information is defined. The load part that contains the information is related to the mean inter-arrival time between new users and the phase duration. Here, the number of concurrent users is defined. For instance, for generating a load of thousand users in one minute

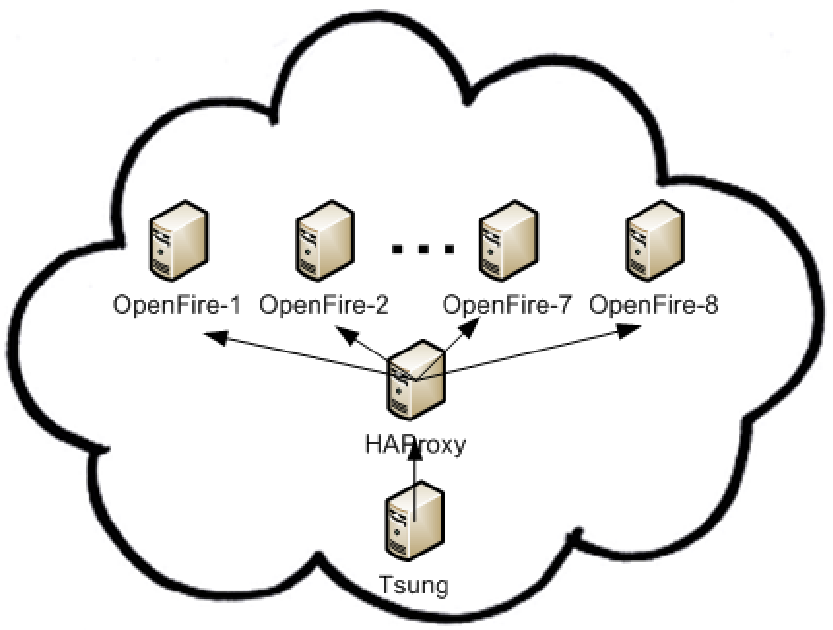


Figure 5.3: Load Test Setup for OpenFire

a mean of 0.06 was used. And finally, the sessions part, in which the testing scenario that is conformed by the client requests is configured.

HAProxy was deployed on a separate machine instance and up to 8 OpenFire worker nodes were setup (Figure 5.3). Servers running on Amazon EC2 infrastructure were using EC2 small instances (a small instance has 1.8 GB of memory). One EC2 computational instance is equivalent to a CPU capacity of 2.66 GHz Intel®Xeon^{TM}processor. Both load balancer and OpenFire nodes were running on 64 bit Linux platforms (Ubuntu). The load balancer was setup for using Round-robin scheduling, so the load can be divided into equal portions among the worker nodes.

The first scenario was considered since XMPP servers can be configured to communicate with each other and a shared database tends to become a bottleneck in situations where high concurrency is required. Tsung can be configured to use usernames and passwords from a range and since it cannot be predicted which server HAProxy directs the connection to, all of the users had to be replicated to every database.

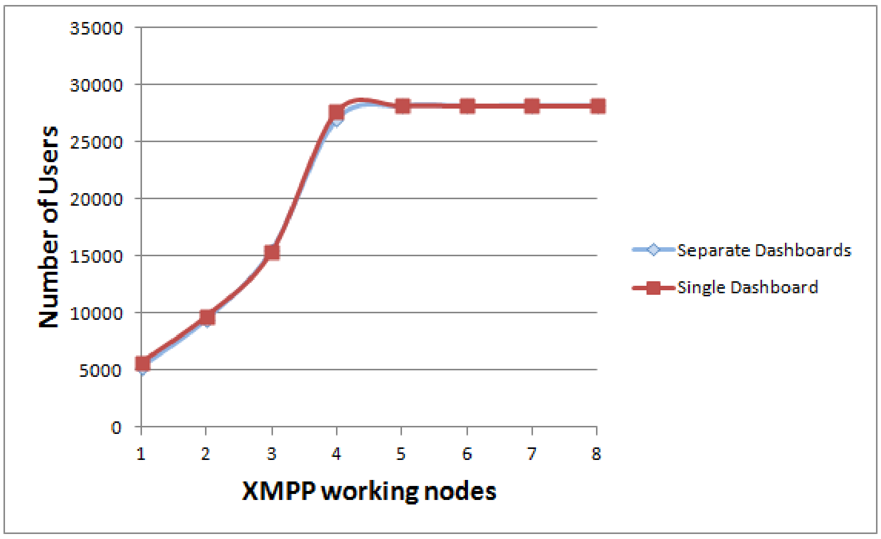


Figure 5.4: OpenFire Load Test Results

In order to determine whether OpenFire server is capable of serving high volume of concurrent users, another scenario was considered, where every XMPP server node was connected to the same database, residing on a separate virtual machine instance. Such an approach simplifies the topology of the architecture and increases data integrity by avoiding data synchronization with other databases. The same users that were created for the first scenario were also used in the second scenario.

Regarding the scenario as the details of the requests, the scenarios were as follows. Every user started a TCP connection with XMPP server. The request was made to HAProxy, which redirected it to appropriate XMPP server. Secondly, XMPP stream headers were exchanged. When the stream was opened, authentication, resource and presence stanzas were exchanged. Once the user was online, it sent a random alphanumeric message to any user online. Thus the tested scenario is the same as the one applied to the Arduino sensor. The results of the tests can be seen in Figure 5.4.

The tests were carried out from 1 to 8 concurrently running XMPP working nodes, altogether 16 test runs. Comparing the two scenarios, the results were almost identical, regarding every number of working nodes. New users were added at the same pace in every tested setting. Therefore the line is a bit concave

Nodes	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Separate Dashboards	5247	9455	15491	27005	28229	28197	28233	28299
Shared Dashboard	5703	9770	15384	27693	28226	28220	28229	28229

Table 5.3: The number of users reached in every node setting

at the beginning, because as new nodes are added the performance of each node increases since the number of new users for a node in a time unit decreases. When six and more instances were running, Tsung and HAProxy became the bottleneck in both of the testing scenarios. The maximum number of concurrently running users were 28233 with the current test environment setting. The numbers can be probably further increased with distributed Tsung instances and more powerful virtual machine deployed for HAProxy instance, but as proof of concept we have shown that OpenFire is highly scalable XMPP server and can be successfully used in live environments. The overall results for every XMPP working node can be seen in Table 5.3.

5.3 Summary

In order to test the reliability of the prototype, battery tests were carried out with both setups - Ethernet and Wireless. Regular 9V battery was used to power the hardware. The initial expectations assumed that the Ethernet setup would definitely last longer that the Wireless. Surprisingly, the test proved us wrong and by a considerable margin. Transporting the data over wire lasted $\approx$ 101 minutes, including the connection negotiation. In contrast, Wireless scenario lasted $\approx$ 162 minutes, a whole hour longer. Although, the WiFly module was configured to use the same amount of power for transmission as the Ethernet Shield, the latter exhausts considerably more power.

The chip can be configured to send sensor data with different transmission power levels (0dBm to +12dBm).

Furthermore, several load tests were carried out to discover the performance of an OpenFire XMPP working node. Two different scenarios were considered - separate dashboards and shared dashboard. The dashboard in both scenarios consisted of a full-scale MySQL database. These two scenarios were taken into consideration to find out whether the dashboard might become the bottleneck of the overall system. The results of the tests proved that the setting does not have any considerable impact on the system and for simplicity and data integrity, it is preferable to configure a shared dashboard. Although, from six and more deployed working nodes the testing environment setting itself became the bottleneck of the test, 28233 concurrent online users were achieved. What is more, to give a meaning to test regarding the prototype, the commands exchanged by every user were exactly the same as in the case of the prototype.

6

Related Work

Although there isnt anything similar published for Arduino, there are some similar proposals published.

An XMPP implementation has been created for Contiki operating system (14). Contiki is an open-source and portable operating system, designed mainly for memory-constrained network systems and embedded systems on microcontrollers. The approach created for Contiki relies on info/query requests. XMPP clients can add the sensor entities to their buddy list and if one wishes to request sensor data, he/she simply sends and iq request over the Internet to the sensor entity in order to obtain the information. Furthermore, the creators have only implemented the bare minimum of XMPP to spare the limited resources of the system. The lightweight, open-source XMPP implementation has been named microXMPP.

Message Queue Telemetry Transport (MQTT) is somewhat similar protocol to XMPP. It supports publish/subscribe technique and instant messaging. An extension of the protocol, MQTT-S, alongside with MQTT has been employed as a data transfer protocol for sensor networks in (18). The consuming applications, in their approach, register as subscribers and the sensors act as publishers. Whenever sensor data is available, the appropriate applications can receive it. In order to manage such data delivery, a message broker, similar to an XMPP server, is set up between the sensors and applications. This way the sensors do not have to maintain several connections to the applications that request the data and save the limited resources and memory of the device. Although, there is a broker between the requester and the sensor, the latter does not directly communicate with the broker. Because the broker does not understand MQTT-S protocol, there is yet another component, gateway, established between the actuators and the broker to translate the protocol to MQTT which can be understood by the broker.

7

Conclusions and Future Research Directions

Pervasive services for mobile users are constrained to the amount of users that can be handled and smart technologies are rather expensive to install and maintain. The thesis thus proposed a solution to counter these problems, aiming to provide mobile users with contextual information about the environment. The implementation takes advantage of Arduino low-cost microcontroller and sensor hardware, real-time XMPP communication protocol and the cloud infrastructure to store the data.

The data read from the sensors is sent to the cloud where it can be further processed for any desired purpose. The cloud was preferred since it offers means for simple application setup and scales on demand, whenever deemed necessary. Furthermore, a virtual machine instance has access to an order of magnitude more powerful resources than Arduino itself.

The prototype was developed to work with two different network interfaces - Ethernet and Wireless. Since the underlying class structure of communication interfaces is the same for all Arduino communication devices, the solution can be easily complied with other means of communication, e.g. Bluetooth, GPRS.

A particular advantage of XMPP over other protocols is that it provides asynchronous communication and is widely used as an IM protocol in several popular instant messaging clients. This gives assurance that the protocol is here to stay and the open-source community is continuously developing it further. Besides the aforementioned, XMPP is designed to grant any kind of custom extensions that comply with XML guidelines. What is more, the implementation demonstrates that the core version of the protocol can be easily implemented on low-cost limited resource devices.

To test the reliability of the prototype, battery tests were carried out with both setups - Ethernet and Wireless. Regular 9V battery was used to power the hardware. The initial expectations assumed that the Ethernet setup would definitely last longer that the Wireless. Surprisingly, the test proved us wrong and by a considerable margin. Transporting the data over wire lasted for 101 minutes, including the connection negotiation. In contrast, Wireless scenario lasted for 162 minutes, a whole hour longer. Although, the WiFly module was configured to use the same amount of power for transmission as Ethernet Shield, the latter is exhausting more power.

Furthermore, several load tests were carried out to discover the performance of an OpenFire XMPP working node. Two different scenarios were considered - separate dashboard for every working node and a shared dashboard for all of the working nodes. The dashboard in both scenarios was implemented as a database. These two scenarios were taken into consideration to find out whether the dashboard might become the bottleneck of the overall system. The results of the tests proved that the setting does not have any considerable impact on the system and for simplicity and data integrity, it is preferable to configure a shared dashboard. Although, from six and more deployed working nodes the testing environment setting itself became the bottleneck of the test, 28233 concurrent online users were achieved. What is more, to give a meaning to test regarding the prototype, the commands exchanged by every user were exactly the same as in the case of the prototype.

Regarding further development of the proposed prototype, there are some ideas. The prototype captures data from the sensors and delivers it to the server at precise intervals as configured in the Arduino sketch. Therefore, the current solution is rather stiff and does not offer dynamics. A rather useful feature is requesting the data on demand, hence enhancing the solution by adding versatility.

Furthermore, the prototype delivers the sensor data to only one user at a time, keeping it static. If the receiving user's credentials change, the whole sketch needs to be re-configured and uploaded to Arduino. Although, this approach is fully functional, its rather cumbersome, time-consuming and does not deplete the possibilities of XMPP.

To overcome the problem explained, publish-subscribe technique can be applied. XMPP has an extension implemented to support such behaviour. In this case the static receiver configuration can be omitted altogether and Arduino instance does not need the knowledge who is at the receiving end of the transmission channel. Thus the information can be published to unlimited amount of entities. Such entities could be simple users or other servers.

Since XMPP is not limited to communicate in only single domain, an XMPP server can communicate to several other servers in different domains altogether. This opportunity provides the means for integrating the sensor server with other servers without the tedious service integration as is necessary with SOAP or REST.

Sensoritelt kogutud keskkonnapõhiste andmete edastamine mobiilsetele kasutajatele toetudes XMPP protokollile

Magistritöö (30 EAP)
Kaarel Hanson
Resümee

Tänapäeval võimaldavad tehnoloogilised edusammud kasutaja käitumise seiremeetodites automatiseerida teatud arvutusülesandeid, mis täidetakse kasutaja kavatsust prognoosides. Nutitelefonid rikastavad mobiilseid rakendusi prognoosiva käitumisega kasutatavuses, mis võimaldab käitumismustritega sammu pidada. Üldiselt on sellist käitumist võimalik saavutada nutitelefoni enda vahenditega, kasutades telefoni sisse ehitatud mikromehaanilisi seadmeid, mis võimaldavad keskkonda tajuda. Lisaks võivad mobiilsed rakendused kasutaja mobiilse kogemuse rikastamiseks lõigata kasu keskkonda integreeritud hajuslausteenustest, nagu näiteks ümbrustundlike mängude, kodu automatiseerimise tarkvara jms puhul.

Ent mobiilikasutajatele suunatud lausteenuseid pakkuvatel elektroonilistel seadmetel, mis koguvad sensorite abi keskkonnast informatsiooni, on teatavad riistvaralised piirangud (arvutusjõudlus, mälu, salvestusmeedia, energiatarbimine jne).

Seega, lausüsteemid ei ole võimelised nõudluse suurenemisel skaleeruma ega rakendama suurt arvutusjõudlust.

Töö eesmärgiks on pakkuda lahendus ületamaks skaleeruvuse, andmete terviklikkuse säilitamise ja vähese arvutusjõudluse probleeme ning rikastada nutitelefoni rakendusi detailsete kasutajapõhiste andmetega. Eesmärgi saavutamiseks transporditakse sensoritelt kogutud informatsioon optimiseeritud XMPP protokolli abiga Arduino mikrokontrollerist pilvesüsteemi. Süsteemi ehitamiseks kasutatakse Arduino poolt pakutavat odavat riistvara, samas kui pilvesüsteemi usaldusväärset ja kõrge kättesaadavusega vahendeid kasutatakse mikrokontrollerist saadetud andmete salvestamiseks ja edaspidiseks töötlemiseks.

Töö käigus testiti mikrokontrolleri energia nõudlust, kasutades 9V patareid, nii juhtme kui ka juhtmevaba liidesega. Tulemused tõestasid eeldustele vastupidiselt, et juhtmevaba süsteemi energia nõudlus on suurem. Lisaks testiti vabavara XMPP serveri jõudlust pilvesüsteemi keskkonnas ning tulemused näitasid, et XMPP võimaldab üheaegselt serveerida suure hulga kasutajaid.

Bibliography

[1] RFC6120, Extensible Messaging and Presence Protocol (XMPP): Core, http://xmpp.org/rfcs/rfc6120.html. 5

[2] Apple Inc, IPhone, http://www.apple.com/iphone/. 6

[3] Google Inc, Android, http://www.android.com/. 6, 9

[4] H. Flores, S. N. Srirama, C. Paniagua, A Generic Middleware Framework for Handling Process Intensive Hybrid Cloud Services from Mobiles, in: The 9th International Conference on Advances in Mobile Computing & Multimedia (MoMM-2011), ACM, 2011, pp. 87–95. 7

[5] S. Srirama, H. Flores, C. Paniagua, Zompopo: Mobile Calendar Prediction based on Human Activities Recognition using the Accelerometer and Cloud Services, in: Next Generation Mobile Applications, Services and Technologies (NGMAST), 2011 5th International Conference on, IEEE, 2011, pp. 63–69. 7

[6] H. Flores, S. N. Srirama, MCM: Mobile Cloud Middleware, submitted for publication in IEEE transactions on mobile computing (2012). 8

[7] Amazon, Inc, Amazon - Amazon Web Services, http://aws.amazon.com/. 12

[8] Google Inc., Google Code - google application engine, http://code.google.com/appengine/ (2011). 12

[9] D. Bernstein, D. Vij, Using XMPP as a transport in Intercloud Protocols, DOI= http://www. cloudstrategy-partners. com/6. html. 15

[10] J. Wagener, O. Spjuth, E. Willighagen, J. Wikberg, XMPP for cloud computing in bioinformatics supporting discovery and invocation of asynchronous web services, BMC bioinformatics 10 (1) (2009) 279. 16

[11] Arduino Inc., Arduino, http://www.arduino.cc. 16

[12] WIZnet, W5100 Datasheet, http://www1.futureelectronics.com/doc/WIZNET 21

[13] Roving Networks, RN-XV Wireless Module Datasheet, http://www.rovingnetworks.com/. 22

[14] A. Hornsby, E. Bail, μxmpp: Lightweight implementation for low power operating system contiki, in: Ultra Modern Telecommunications & Workshops, 2009. ICUMT'09. International Conference on, IEEE, 2009, pp. 1–5. 24, 58

[15] RFC4422, Simple Authentication and Security Layer (SASL), http://www.ietf.org/rfc/rfc4422.txt. 36

[16] RFC4616, The PLAIN Simple Authentication and Security Layer (SASL) Mechanism, http://www.ietf.org/rfc/rfc4616.txt. 36

[17] Energizer, Energizer 522 9V Battery Datasheet, http://data.energizer.com/PDFs/522.pdf. 50

[18] U. Hunkeler, H. Truong, A. Stanford-Clark, MQTT-SA publish/subscribe protocol for Wireless Sensor Networks, in: Communication Systems Software and Middleware and Workshops, 2008. COMSWARE 2008. 3rd International Conference on, IEEE, 2008, pp. 791–798. 58