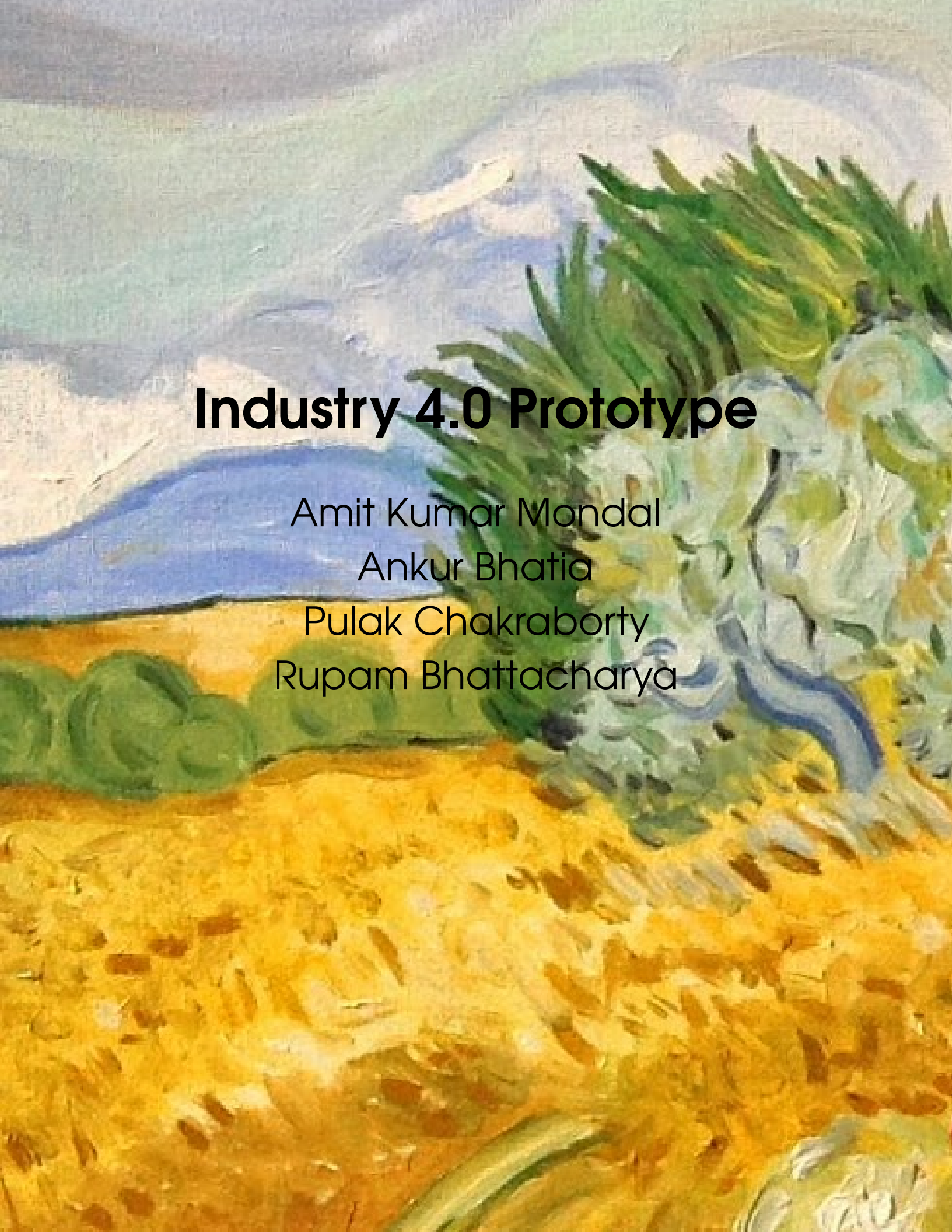


Industry 4.0 Prototype

Amit Kumar Mondal
Ankur Bhatia
Pulak Chakraborty
Rupam Bhattacharya

INTERDISCIPLINARY PROJECT, TECHNICAL UNIVERSITY OF MUNICH

2 December 2015

Table of Contents

1. Introduction

1.1 Motivation

The Internet of Things is indeed the talk of the town these days. From the perspective of the industries, it makes them more efficient by enabling us to control them and get their state from virtually anywhere. The biggest motivation for us was to be a part of this industrial revolution and go back with some learning that we can apply later on in our careers as well. As mentioned, being a very new topic, there was a lot of scope for research and an opportunity to come up with a prototype that has not been developed by many yet. It gives us immense pleasure and a sense of satisfaction, that we gave our 100% and developed something meaningful that can be used as a starting point to develop an end to end IoT solution in the actual production environment.

2. Scalable Architecture

2.1 Tools

1. Raspberry Pi

2.2 Technologies

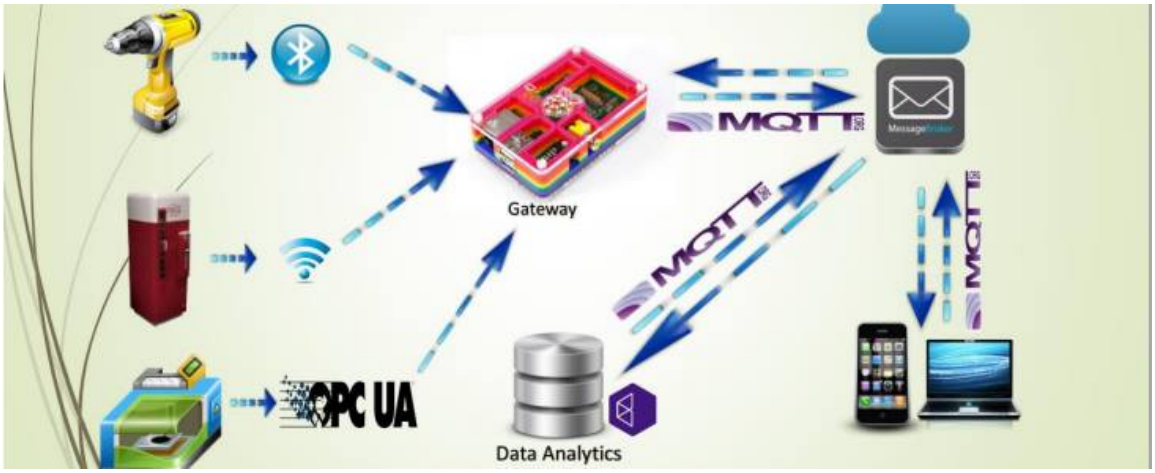
The following tools were required to develop the end to end solution of our Interdisciplinary Project.

1. Java
2. OSGi
3. Eclipse Kura
4. Javascript
5. AngularJS
6. Python
7. MongoDB
8. Java
9. Crodova
10. Google Protocol Buffer

The following protocols were incorporated due to projects requirements as well as to make the base architecture scalable.

1. MQTT
2. Bluetooth
3. Socket NIO - WiFi
4. OPC-UA

2.3 Overall Architecture



2.4 MQTT

The primary mechanism that devices and applications use to communicate with the IBM Internet of Things Foundation is MQTT; this is a protocol designed for the efficient exchange of real-time data with sensor and mobile devices.

MQTT runs over TCP/IP and, while it is possible to code directly to TCP/IP, you might prefer to use a library that handles the details of the MQTT protocol for you. You will find there's a wide range of MQTT client libraries available at mqtt.org, with the best place to start looking being the Eclipse Paho project. IBM contributes to the development and support of many of these libraries.

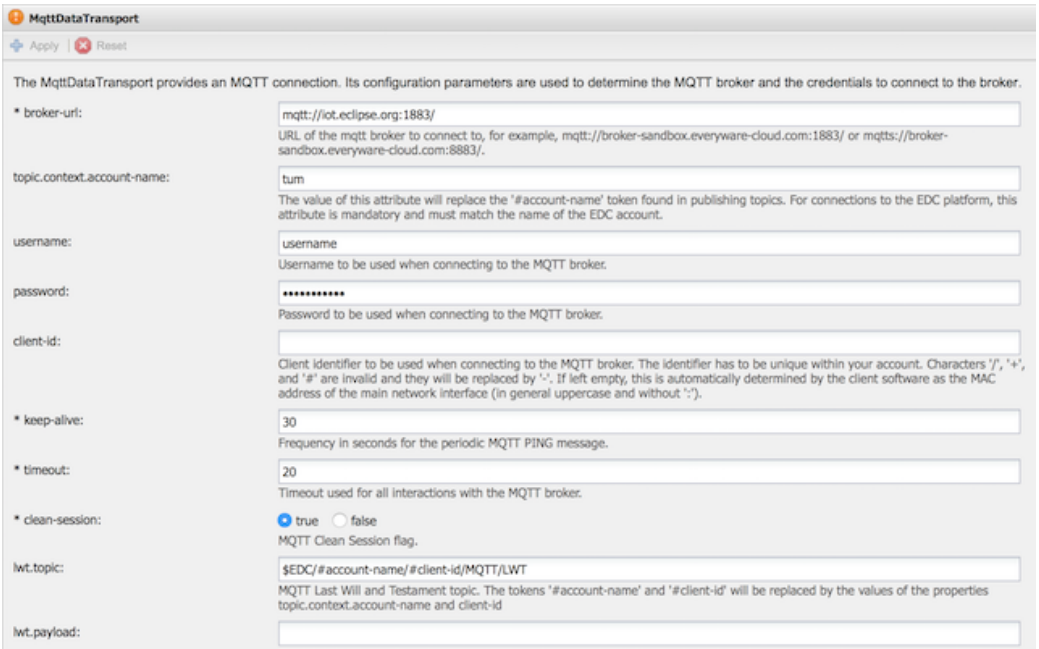
MQTT 3.1 is the version of the protocol that is in widest use today. Version 3.1.1 contains a number of minor enhancements, and has been ratified as an OASIS Standard.

One reason for using version 3.1.1 is that the maximum length of the MQTT Client Identifier (Client ID) is increased from the 23 character limit imposed by 3.1. The IoT service will often require longer Client ID's and will accept long Client ID's with either version of the protocol however some 3.1 client libraries check the Client ID and enforce the 23 characters limit.

Every registered organization has a unique endpoint which must be used when connecting MQTT clients for applications and devices in that organization.

The MQTT protocol provides three qualities of service for delivering messages between clients and servers: "at most once", "at least once" and "exactly once". Events and commands can be sent using any quality of service level, however you should carefully consider whether what the right level is for your needs. It is not a simple case that QoS2 is "better" than QoS0.

The following image shows the various options to configure MQTT in Gateway.



2.5 Java - OSGi

OSGi reduces complexity by providing a modular architecture for today's large-scale distributed systems as well as small, embedded applications. Building systems from in-house and off-the-shelf modules significantly reduces complexity and thus development and maintenance expenses. The OSGi programming model realizes the promise of component-based systems.

The key reason OSGi technology is so successful is that it provides a very mature component system that actually works in a surprising number of environments. The OSGi component system is actually used to build highly complex applications like IDEs (Eclipse), application servers (GlassFish, IBM Websphere, Oracle/BEA Weblogic, Jonas, JBoss), application frameworks (Spring, Guice), industrial automation, residential gateways, phones, and so much more.

So, what benefits does OSGi's component system provide you?

Reduced Complexity – Developing with OSGi technology means developing bundles: the OSGi components. Bundles are modules. They hide their internals from other bundles and communicate through well defined services. Hiding internals means more freedom to change later. This not only reduces the number of bugs, it also makes bundles simpler to develop because correctly sized bundles implement a piece of functionality through well defined interfaces.

Reuse – The OSGi component model makes it very easy to use many third party components in an application. An increasing number of open source projects provide their JARs ready made for OSGi. However, commercial libraries are also becoming available as ready made bundles.

Real World – The OSGi framework is dynamic. It can update bundles on the fly and services can come and go. Developers used to more traditional Java see this as a very problematic feature and fail to see the advantage. However, it turns out that the real world is highly dynamic and having dynamic services that can come and go makes the services a perfect match for many real world scenarios. For example, a service could model a device in the network. If the device is detected, the service is registered. If the device goes away, the service is unregistered. There are a surprising number of real world scenarios that match this dynamic service model. Applications can therefore reuse the powerful primitives of the service registry (register, get, list with an expressive filter language, and waiting for services to appear and disappear) in their own domain. This not only saves writing code, it also provides global visibility, debugging tools, and more functionality than would have implemented for a dedicated solution. Writing code in such a dynamic environment sounds like a nightmare, but fortunately, there are support classes and frameworks that take most, if not all, of the pain out of it.

Easy Deployment – The OSGi technology is not just a standard for components. It also specifies how components are installed and managed. This API has been used by many bundles to provide a management agent. This management agent can be as simple as a command shell, a TR-69 management protocol driver, OMA DM protocol driver, a cloud computing interface for Amazon's EC2, or an IBM Tivoli management system. The standardized management API makes it very easy to integrate OSGi technology in existing and future systems.

Dynamic Updates – The OSGi component model is a dynamic model. Bundles can be installed, started, stopped, updated, and uninstalled without bringing down the whole system. Many Java developers do not believe this can be done reliably and therefore initially do not use this in production. However, after using this in development for some time, most start to realize that it actually works

and significantly reduces deployment times.

Adaptive – The OSGi component model is designed from the ground up to allow the mixing and matching of components. This requires that the dependencies of components need to be specified and it requires components to live in an environment where their optional dependencies are not always available. The OSGi service registry is a dynamic registry where bundles can register, get, and listen to services. This dynamic service model allows bundles to find out what capabilities are available on the system and adapt the functionality they can provide. This makes code more flexible and resilient to changes.

Transparency – Bundles and services are first class citizens in the OSGi environment. The management API provides access to the internal state of a bundle as well as how it is connected to other bundles. For example, most frameworks provide a command shell that shows this internal state. Parts of the applications can be stopped to debug a certain problem, or diagnostic bundles can be brought in. Instead of staring at millions of lines of logging output and long reboot times, OSGi applications can often be debugged with a live command shell.

Versioning – OSGi technology solves JAR hell. JAR hell is the problem that library A works with library B;version=2, but library C can only work with B;version=3. In standard Java, you're out of luck. In the OSGi environment, all bundles are carefully versioned and only bundles that can collaborate are wired together in the same class space. This allows both bundle A and C to function with their own library. Though it is not advised to design systems with this versioning issue, it can be a life saver in some cases.

Simple – Using OSGi is surprisingly simple, despite the powerful dependency management, configuration, and dynamics, OSGi code looks almost identical to classic Java code. A number of easy-to-use annotations tell the runtime how a particular class wants to use the dynamics, configuration, and dependencies on other services. The defaults completely hide the dynamics and OSGi. This very simple model allows the gradual use of more advanced features.

Small – The OSGi Release 4 Framework can be implemented in about a 300KB JAR file. This is a small overhead for the amount of functionality that is added to an application by including OSGi. OSGi therefore runs on a large range of devices: from very small, to small, to mainframes. It only asks for a minimal Java VM to run and adds very little on top of it.

Fast – One of the primary responsibilities of the OSGi framework is loading the classes from bundles. In traditional Java, the JARs are completely visible and placed on a linear list. Searching a class requires searching through this (often very long, 150 is not uncommon) list. In contrast, OSGi pre-wires bundles and knows for each bundle exactly which bundle provides the class. This lack of searching is a significant speed up factor at startup.

Lazy – Lazy in software is good and the OSGi technology has many mechanisms in place to do things only when they are really needed. For examples, bundles can be started eagerly, but they can also be configured to only start when another bundle is using them. Services can be registered but only created when they are used. The specifications have been optimized several times to allow for these kind of lazy scenarios that can save tremendous runtime costs.

Secure – Java has a very powerful fine grained security model at the bottom but it has turned out very hard to configure in practice. The result is that most secure Java applications are running with a binary choice: no security or very limited capabilities. The OSGi security model leverages the

fine grained security model but improves the usability (as well as hardening the original model) by having the bundle developer specify the requested security details in an easily audited form while the operator of the environment remains fully in charge. Overall, OSGi likely provides one of the most secure application environments that is still usable short of hardware protected computing platforms.

Humble – Many frameworks take over the whole VM, they only allow one instance to run in a VM. The flexibility of the OSGi specifications is demonstrated by how it can even run inside a J2EE Application Server. Many developers wanted to run OSGi but their companies did not allow them to deploy normal JARs. Instead, they included an OSGi framework in their WAR file and loaded their bundles from the file system or over the network. OSGi is so flexible that one application server can easily host multiple OSGi frameworks.

Non Intrusive – Applications (bundles) in an OSGi environment are left to their own. They can use virtually any facility of the VM without the OSGi restricting them. Best practice in OSGi is to write Plain Old Java Objects and for this reason, there is no special interface required for OSGi services, even a Java String object can act as an OSGi service. This strategy makes application code easier to port to another environment.

Runs Everywhere – Well, that depends. The original goal of Java was to run anywhere. Obviously, it is not possible to run all code everywhere because the capabilities of the Java VMs differ. A VM in a mobile phone will likely not support the same libraries as an IBM mainframe running a banking application. There are two issue to take care of. First, the OSGi APIs should not use classes that are not available on all environments. Second, a bundle should not start if it contains code that is not available in the execution environment. Both of these issues have been taken care of in the OSGi specifications.

2.6 Eclipse Kura

Eurotech, a leading supplier of embedded technologies, products and systems, announced its role and contribution to Eclipse Kura, an open source incubator project aimed at providing an OSGi-based container for M2M applications running in service gateways.

Deploying and configuring one device to act as a node in the Internet of Things is relatively easy. Doing the same for hundreds or thousands of devices, supporting several local applications, is not so easy though. This is where the new Eclipse project Kura comes in. Kura offers a platform that can live at the boundary between the private device network and the local network, public Internet or cellular network, providing a manageable and intelligent gateway for that boundary capable of running applications that can harvest locally gathered information and deliver it reliably to the cloud.

Through the Kura project, Eurotech will provide a set of common services for Java developers building M2M applications, including I/O access, data services, network configuration and remote management.

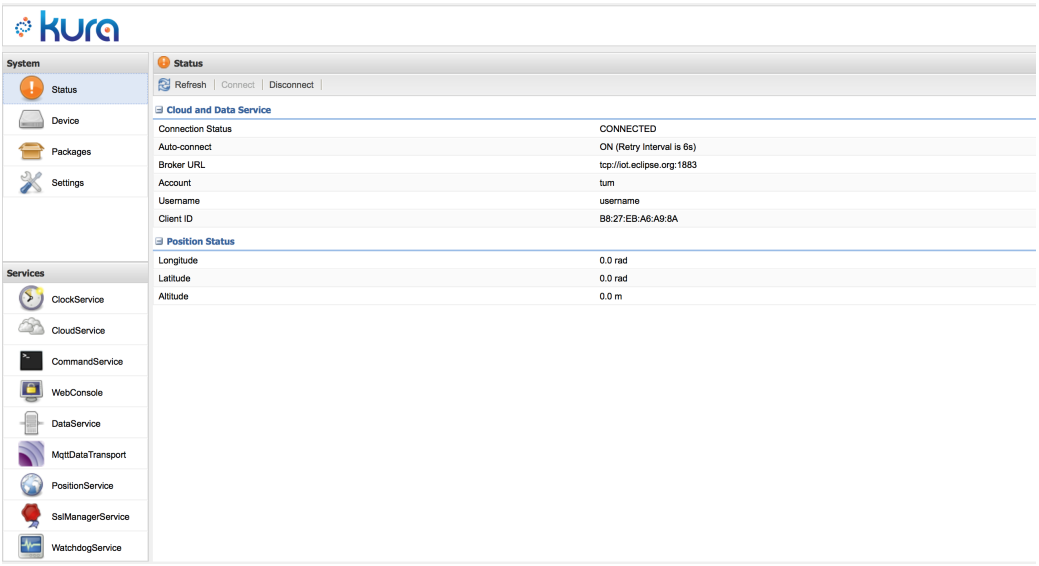
Developers working with Kura will find that, as it is within an OSGi (Open Service Gateway initiative) container, they will be working with a standard framework for handling events, packaging code and a range of standard services. An application in Kura is delivered as an OSGi module and is run within the container along with the other components of Kura. Using the Eclipse Paho MQTT library, Kura provides a store-and-forward repository service for those applications to take the information gathered from the locally attached devices or network-attached devices sending that data onwards to MQTT brokers and other cloud services.

Applications can be remotely deployed as OSGi bundles and their configuration imported (or exported) through a snapshot service. The same configuration service can be used to setup Kura's other OSGi compatible services – DHCP, DNS, firewall, routing and WiFi – which can be used to manage the networking setup of a gateway and provision private LANs and WLANs. Other bundled services available include: position, a GPS location service to geolocate your gateways; click, a time service to ensure good time synchronization; db, a database service for local storage using a embedded SQL database and process and watchdog services to keep things running smoothly.

To talk to network attached devices, applications can use Java's own networking capabilities (or optional support field protocols, such as Modbus and CAN Bus) to plug into existing device infrastructure. By abstracting the hardware using OSGi services for Serial, USB and Bluetooth communications, Kura gives application developers portable access to a wide range of common devices though they can still use Java's own range of communications APIs when appropriate. An API for devices attached via GPIO, I2C, PWM or SPI will allow a system integrator to incorporate custom hardware as part of their gateway.

The Kura package is completed with a web front end which allows the developer or administrator to remotely log in and configure all the OSGi-compliant bundles and which developers can utilize to provide a web facing aspect to their own application's configuration needs. Developers should find that the Eclipse IDE's OSGi tooling makes the route from code conception to installation on Kura an easily navigable path too.

Kura Web UI


kura
System
Status
Device
Packages
Settings
Services
ClockService
CloudService
CommandService
WebConsole
DataService
MqttDataTransport
PositionService
SslManagerService
WatchdogService
Status
Refresh
Connect
Disconnect
Cloud and Data Service
Connection Status
CONNECTED
Auto-connect
ON (Retry Interval is 6s)
Broker URL
tcp://iot.eclipse.org:1883
Account
tum
Username
username
Client ID
B8:27:EB:A6:A9:8A
Position Status
Longitude
0.0 rad
Latitude
0.0 rad
Altitude
0.0 m

2.7 Cordova

Apache Cordova (formerly PhoneGap) is a popular mobile application development framework originally created by Nitobi. Adobe Systems purchased Nitobi in 2011, rebranded it as PhoneGap, and later released an open source version of the software called Apache Cordova. Apache Cordova enables software programmers to build applications for mobile devices using JavaScript, HTML5, and CSS3, instead of relying on platform-specific APIs like those in Android, iOS, or Windows Phone. It enables wrapping up of CSS, HTML, and JavaScript code depending upon the platform of the device. It extends the features of HTML and JavaScript to work with the device. The resulting applications are hybrid, meaning that they are neither truly native mobile application (because all layout rendering is done via Web views instead of the platform's native UI framework) nor purely Web-based (because they are not just Web apps, but are packaged as apps for distribution and have access to native device APIs). Mixing native and hybrid code snippets has been possible since version 1.9.

2.8 Google Protocol Buffer

Service-Oriented Architecture has a well-deserved reputation amongst Ruby and Rails developers as a solid approach to easing painful growth by extracting concerns from large applications. These new, smaller services typically still use Rails or Sinatra, and use JSON to communicate over HTTP. Though JSON has many obvious advantages as a data interchange format - it is human readable, well understood, and typically performs well - it also has its issues.

Where browsers and JavaScript are not consuming the data directly – particularly in the case of internal services – it's our opinion that structured formats, such as Google's Protocol Buffers, are a better choice than JSON for encoding data.

Google developed Protocol Buffers for use in their internal services. It is a binary encoding format that allows you to specify a schema for your data using a specification language

You can package messages within namespaces or declare them at the top level as above. The snippet defines the schema for a Person data type that has three fields: id, name, and email. In addition to naming a field, you can provide a type that will determine how the data is encoded and sent over the wire - above we see an int32 type and a string type. Keywords for validation and structure are also provided (required and optional above), and fields are numbered, which aids in backward compatibility, which we'll cover in more detail below.

The Protocol Buffers specification is implemented in various languages: Java, C, Go, etc. are all supported, and most modern languages have an implementation if you look around. Ruby is no exception and there are a few different Gems that can be used to encode and decode data using Protocol Buffers. What this means is that one spec can be used to transfer data between systems regardless of their implementation language.

There is a certain painful irony to the fact that we carefully craft our data models inside our databases, maintain layers of code to keep these data models in check, and then allow all of that forethought to fly out the window when we want to send that data over the wire to another service. All too often we rely on inconsistent code at the boundaries between our systems that don't enforce the structural components of our data that are so important. Encoding the semantics of your business objects once, in proto format, is enough to help ensure that the signal doesn't get lost between applications, and that the boundaries you create enforce your business rules.

Numbered fields in proto definitions obviate the need for version checks which is one of the explicitly stated motivations for the design and implementation of Protocol Buffers. As the developer documentation states, the protocol was designed in part to avoid "ugly code"

In addition to explicit version checks and the lack of backward compatibility, JSON endpoints in HTTP based services typically rely on hand-written ad-hoc boilerplate code to handle the encoding and decoding of Ruby objects to and from JSON. Parser and Presenter classes often contain hidden business logic and expose the fragile nature of hand parsing each new data type when a stub class as generated by Protocol Buffers (that you generally never have to touch) can provide much of the same functionality without all of the headaches. As your schema evolves so too will your proto generated classes (once you regenerate them, admittedly), leaving more room for you to focus on the challenges of keeping your application going and building your product.

Because Protocol Buffers are implemented in a variety of languages, they make interoperability

between polyglot applications in your architecture that much simpler. If you're introducing a new service with one in Java or Go, or even communicating with a backend written in Node, or Clojure, or Scala, you simply have to hand the proto file to the code generator written in the target language and you have some nice guarantees about the safety and interoperability between those architectures. The finer points of platform specific data types should be handled for you in the target language implementation, and you can get back to focusing on the hard parts of your problem instead of matching up fields and data types in your ad hoc JSON encoding and decoding schemes.

Protocol Buffers offer several compelling advantages over JSON for sending data over the wire between internal services. While not a wholesale replacement for JSON, especially for services which are directly consumed by a web browser, Protocol Buffers offers very real advantages not only in the ways outlined above, but also typically in terms of speed of encoding and decoding, size of the data on the wire, and more.

2.9 MongoDB

IoT means parsing mass amounts of data. Throwing hardware at the issue is financially problematic, because often the data surges are seasonal

With the proliferation of new data sources, from sensors to smart devices, your IoT system must be able to adapt flexibly

Most IoT apps are deployed in a cloud environment. Companies need software that can manage global scenarios

It's not about collecting the data. It's about deriving real world insight. That means real-time analysis.

3. Protocols incorporated in Gateway

3.1 Bluetooth using Java

Bluetooth is a low-cost, short-range wireless technology that has become popular among those who want to create personal area networks (PANs). Each PAN is a dynamically created network built around an individual, that enables devices such as cellular phones and personal digital assistants (PDAs) to connect automatically and share data immediately. To support development of Bluetooth-enabled software on the Java platform, the Java Community Process (JCP) has defined JSR 82, the Java APIs for Bluetooth Wireless Technology (JABWT).

Bluetooth radio technology is based on the 2.45 GHz Industrial, Scientific, and Medical (ISM) frequency band, which is unlicensed and globally available. When Bluetooth devices connect to each other, they form a piconet, a dynamically created network that comprises a master device and up to seven slave devices. Bluetooth also supports connections between piconets: When a master on one piconet becomes a slave on another, it provides a bridge between them.

The Bluetooth protocol stack provides a number of higher-level protocols and APIs for service discovery and serial I/O emulation, and a lower-level protocol for packet segmentation and reassembly, protocol multiplexing, and quality of service.

Bluetooth interoperability profiles – not to be confused with J2ME profiles – describe cross-platform interoperability and consistency requirements. They include the Generic Access Profile that defines device-management functionality, the Service Discover Application Profile that defines the aspects of service discovery, and the Serial Port Profile that defines the interoperability requirements and capabilities for serial cable emulation. You can learn about these and other profiles in the Bluetooth specification.

JSR 82 exposes the Bluetooth software stack to developers working on the Java platform. Of special interest are the Service Discovery Protocol (SDP), the Serial Port Profile RFCOMM for serial emulation, and the Logical Link Control and Adaptation Profile (L2CAP), which provides

connection-oriented data services to upper-layer protocols such as segmentation and reassembly operation, and protocol multiplexing. Note that JABWT doesn't support connectionless L2CAP.

A Bluetooth-enabled application can be either a server or a client – a producer of services or a consumer – or it can behave as a true peer-to-peer endpoint by exposing both server and client behavior.

We have a scenario when the Bluetooth RFCOMM Server is connected to a Bluetooth RFCOMM client, the server starts broadcasting data continuously and the client needs to consume the data in real-time for further usage. We have found a drawback in Java Bluetooth Communication which restricts the communication to occur in a **non-blocking** way because the stream connection returned from a successful pairing of devices, is Input Stream which operates in a blocking way. **It doesn't support any mechanism to incorporate non-blocking IO**.

For blocking IO communication, the Java mechanism that we have incorporated in the gateway works perfectly but due to our situation of real-time data collection, it failed. Still we have thought not to remove the functionality from the Gateway so that it can further be extended for non-blocking communication.

The responsible plugins:
1. de.tum.in.bluecove
2. de.tum.in.bluecove.fragment
3. de.tum.in.bluetooth
4. de.tum.in.bluetooth.milling.machine

3.2 Bluetooth using Python

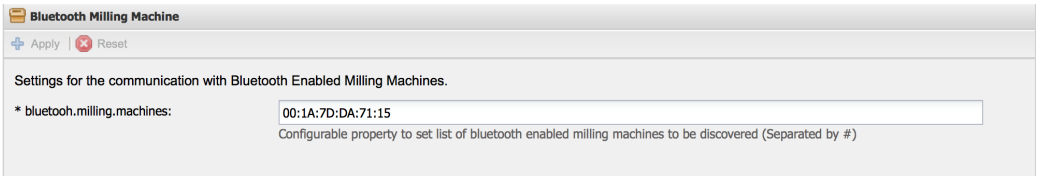
Bluetooth Python extension module to allow Python ” “developers to use system Bluetooth resources. PyBluez works ” “with GNU/Linux and Windows XP.

As pyBluez is newer than the APIs provided by Java JSR 82, it does support non-blocking IO mechanism. That’s why we finally had to build the solution using python and got it integrated with Java in a modular way.

The responsible plugin:

1. de.tum.in.bluetooth.milling-machine

The following image shows the variegated options to configure Bluetooth in Gateway.



3.3 WiFi Communication - Socket NIO

Nonblocking sockets, introduced in Java 2 Standard Edition 1.4, allow net communication between applications without blocking the processes using the sockets. In our context of real-time data collection through WiFi, we had to incorporate Nonblocking sockets.

As of Java 1.4, a programmer can use a brand-new set of APIs for I/O operations. This is the result of JSR 51, which started in January 2000 and has been available to programmers since Java 1.4 beta. Some of the most important new features in Java 1.4 deal with subjects such as high performance read/write operations on both files and sockets, regular expressions, decoding/encoding character sets, in-memory mapping, and locking files.

A nonblocking socket allows input/output operation on a channel without blocking the processes using it. We are talking about asynchronous high-performance read/write operations that turn upside-down the techniques for designing and developing socked-based applications.

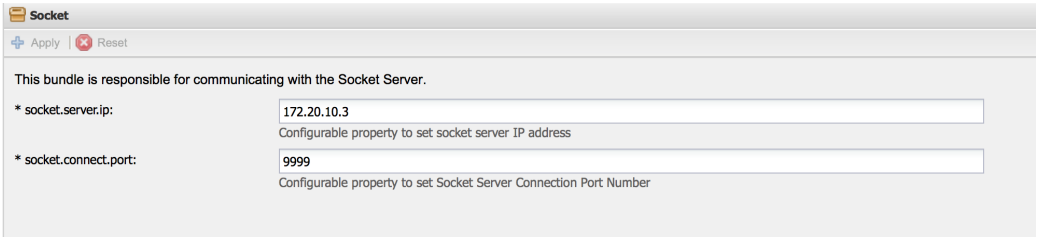
Java developers might ask: why introducing a new technology to handle sockets? What's wrong with the Java 1.3.x sockets? Suppose you would like to implement a server accepting diverse client connections. Suppose, as well, that you would like the server to be able to process multiple requests simultaneously. Using Java 1.3.x, you have two choices to develop such a server:

Implement a multithread server that manually handles a thread for each connection. Using an external third-party module. Both solutions work, but adopting the first one – the whole thread-management solution, with related concurrency and conflict troubles – has to be developed by programmer. The second solution may cost money, and it makes the application dependent on a non-JDK external module. By means of the nonblocking socket, you can implement a nonblocking server without directly managing threads or resorting to external modules.

The responsible plugin:

1. de.tum.in.socket.client

The following image shows the variegated options to configure WiFi Communication in Gateway.



3.4 OPC-UA

OPC UA specifies an abstract set of services and the mapping to a concrete technology. OPC UA does not specify an API but only the message formats for data exchanged on the wire. A communication stack is used on client- and server-side to encode and decode message requests and responses. Different communication stacks can work together as long as they use the same technology mapping.

An OPC UA client consists of a Client Implementation using an OPC UA communication stack. The Client Implementation accesses the communication stack using the OPC UA API. Note that the API is not standardized. It may vary for different programming languages and potentially for different communication stacks. Several communication stacks may exist for different operating systems, programming languages and mappings. For example, there may be a communication stack for Java and a communication stack for Microsoft's new Windows Communication Foundation (WCF). The client-side communication stack allows the client to create re- quest messages based on the service definitions. The client-side communication stack communicates with a server-side communication stack. The OPC Foundation standardized only this communication. Thus, everybody can develop his or her own communication stack with its own API as well.

The server-side communication stack delivers the request messages to the Server Implementation via the OPC UA API. Since the OPC UA API realizes the abstract service specifications, it may be the same as on the client-side. The Server Implementation implements the logic needed to return the appropriate response message.

OPC-UA currently defines two mappings: UA Web Services and UA Native. The first mapping uses SOAP and the various WS-* specifications. The second mapping uses only a simple binary network protocol and integrates TLS-like security mechanisms.

The encoding of the data can be done in XML or UA Binary. UA Binary specifies the serialization of data into a byte string. The UA Binary encoding is faster than the XML encoding since the message size is smaller than for the XML encoding. On the other hand, the XML encoding allows generic SOAP-clients to interpret the data in the SOAP message, while they would only get a binary string using the UA Binary encoding. In theory, the en- coding of the data is independent of the mapping. How- ever, the XML encoding will typically only be used in the UA Web Service mapping in combination with the various WS*-specifications.

The protocol of the UA Web Service mapping is SOAP/HTTP(S) while the UA Native mapping typically runs directly on TCP/IP. For bypassing firewalls, the UA Native mapping also allows putting the binary encoded messages into SOAP messages using HTTP(S).

Of course, UA messages can also be encoded and transported with other protocols, for example using WCF Binary as shown in Figure 1. However, by using this technology you are losing interoperability since you only can talk to WCF clients as long as you do not provide another mapping. Since the described architecture separates the Client and Server Implementation from the communication stack, this is easy to realize.

OPC Unified Architecture (UA) servers offer a step in the direction of Big Data by facilitating efficient SCADA device data management. In automation applications, Big Data means ever-increasing amounts of sensor data from the production process that needs to be effectively managed.

Modern SCADA Systems can communicate directly with an OPC server which in turn communicates with PLCs, RTUs or remote I/O units to pass sensor readings and control signals back and forth between the OPC server and devices.

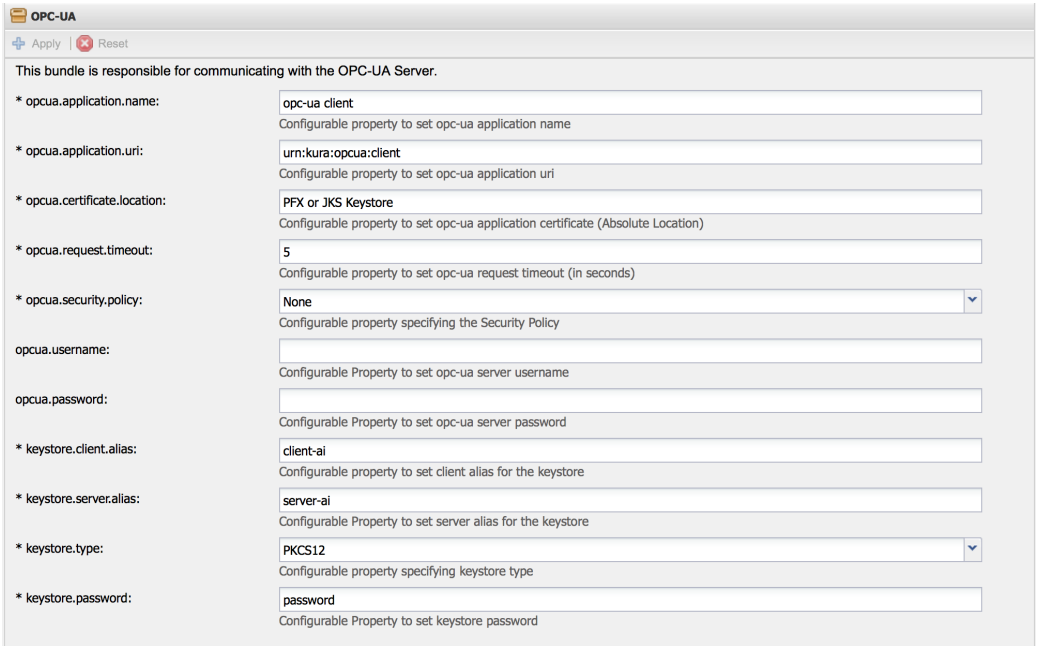
Where the more traditional OPC DA server uses a polling method which can use network bandwidth, newer OPC UA servers use a “report by exception” methodology to reduce the amount of information that the OPC server needs to send to the SCADA software. The combination of OPC UA with Active OPC server technology provides users with a seamless communication solution that can save an impressive amount of bandwidth usage.

In our project context, apart from real-time data collection, we had a requirement to manage the device operations remotely and that’s why we incorporated such protocol to operate on a simulated device which manages switching operations on a LED.

The responsible plugins:

1. de.tum.in.opcua.client
2. de.tum.in.opcua.client.read
3. de.tum.in.opcua.client.read.node
4. de.tum.in.opcua.client.write
5. de.tum.in.opcua.client.write.node
6. de.tum.in.opcua-sdk
7. de.tum.in.led.controller.application

The following image shows the variegated options to configure OPC-UA in Gateway.



4. Protocols incorporated in Mobile Client

4.1 Bluetooth

The communication between the bluetooth device happens via the PUB/SUB protocol. For the same, we have used the broker iot.eclipse.org. For the Bluetooth implementation we have used the Eclipse Paho javascript library. The official documentation for the Library can be found here : https://eclipse.org/paho/clients/js/ .

MQTT Implementation for the Bluetooth. As soon as the Blueooth page is opened in the application, the following three events take place:

1. Connection to the MQTT broker
2. Publishing on a predetermined topic to send a request to get data
3. Subscription on a topic to get Data

This function connects us to a public broker iot.eclipse.org. Note that we have used the time stamp for the client ID to make sure that that we have a unique client connecting from different devices, else there may be issues.

Secondly, the app publishes on the topic $EDC/tum/B8:27:EB:A6:A9:8A/PY-MILLING-V1/EXEC/start with a payload **mqtt data**. This is the same topic on which the Raspberry Pi controller would have subscribed. As soon as the controller gets this message, it indicates that that the mobile client wants Bluetooth data. After receiving this message, the Controller initiates the Bluetooth handshake and starts publishing the Bluetooth data on the topic : $EDC/tum/BLUETOOTH-V1/00:1A:7D:DA:71:15/data.

Lastly, the mobile client subscribes to the topic: $EDC/tum/BLUETOOTH-V1/00:1A:7D:DA:71:15/data

As soon as the mobile client subscribes to this topic, it now will receive all the messages the controller will publish on this topic. To facilitate this, there is a function onMessageArrived(message). This function is called every time there is a message received on the subscribed topic. We have implemented a 3 second wait time before sending a new message. Though we can achieve more

frequent messages, but it might not be as reliable as this. But in any case, it can be changed and tested in the actual production environment.

For the Bluetooth communication, our first approach was to use the Java MQTT plugin which is based on Eclipse Kura and encodes the message in the Google Protocol Buffer format. The encoding plays a very important role in the security of the message as well. However, the Java Library for the Bluetooth communication between the controller and the Bluetooth devices had a lot of issues. The details for the same can be found on Bluetooth using Java Section. This forced us to use a python based library for the communication between the Bluetooth devices and the controller. Also, due to the lack of actual Bluetooth devices, we had to simulate the Bluetooth devices with another Raspberry Pi. Though the problem of receiving continuous data from the Bluetooth devices was solved by python, it introduced new problem of encoding the data in the Google Protocol Buffer. Since our framework had been designed to perform the tasks of encoding and decoding in Java, the same couldn't be used for Bluetooth as the communication was done in python. To implement the encoding in the Google Protocol Buffer, we would have to write the same from scratch in Python. Due to most of our time getting consumed in designing the Bluetooth simulator, we could not implement the encoding for the data in case of Bluetooth. Hence, we made use of the Eclipse Paho JavaScript library.

The message received in the mobile client had the following format: "ForceX=100, ForceY=200, ForceZ=520" For our mobile client to display this message properly, we had to parse this message to display them in the appropriate components and also to extract the Force Values to generate the graphs for visualizations. After parsing the above string, we could extract the Force Values of 100,200 and 520 and put them in the correct UI components as seen in the screenshot and generate the graph accordingly. The parsing function can be found in the JavaScript file parseString.js. The function first parses the string and puts the Force Values in the respective UI Component.

As you might have read before, there is a configuration of a threshold values of the Forces implemented in the Mobile App. The threshold values are the Force values predicted based on the data analytics of the Forces generated by the machine. In order to improve the life of the machine, the threshold values are the recommended Force Values beyond which the machines should not operate. So there is an alert functionality provided in the app, which would make sure that whenever the force values exceed the Threshold, a small alert next to the respective Force will be shown. As soon as the Force Values are parsed, they are compared with the Threshold Values.

The graphical representation of the force data was done using the library canvasjs.min. The official documentation can be found on http://canvasjs.com/javascript-charts/. We made use of the dynamic line charts for the app.

For the graph implementation in the mobile client, as mentioned we have used a Line Chart provided by the Canvas.js library. There are three separate graphs generated for all the three Forces in X,Y and Z coordinates. The function chart.render() generates the graph and displays it in the respective UI component. Moreover, the graph is dynamic in nature. After every new value received, the new point is appended in the graph.

As mentioned, the graph is generated for the three axis. To change the view from one axis to another, just click on the respective Force Block shown in the figure below. The logic used to implement this is very simple. We are actually generating all the graphs in the same position on the screen. But at a time only one of the graph is visible. So by clicking another Force Block, we are

hiding the other two graphs and showing the one we want to show by setting the visibility parameter accordingly in the function UpdateGraphX(),UpdateGraphY() and UpdateGraphZ().

4.2 WiFi Communication

The implementation for WiFi is a little different and a little more complex than the Bluetooth implementation from the mobile client's perspective. Unlike the Bluetooth, here we are not using the Eclipse Paho JavaScript library, we are rather using a Cordova MQTT plugin that we developed in Java from scratch. Since the To interact with the plugin we also have a JavaScript file mqttplugin.js that acts as an interface between the HTML and the Java client. More details about this file will be present later. From the mobile client's perspective, as soon as we open the app's page for WiFi, the app immediately subscribes on a topic $EDC/tum/B8:27:EB:A6:A9:8A/SOCKET-V1/data. However, as mentioned above, these functions are not from the Eclipse Paho Library. We rather call a JavaScript function that in turns call the respective java function fir subscribe. The controller on the other hand publishes the Wifi data on this topic which is then received by the mobile client.

To facilitate the broker mechanism in WiFi along with Google Protocol Buffer, we implemented a Java MQTT plugin for Cordova. You can find this implementation in the /platforms/android/src. The most important file here for the developer is the MQTTplugin.java. In this file, you can see all the relevant functions required for the MQTT.

The parameters for the MQTT connection can be configured from the Settings page. Once the parameters have been provided, those are stored in the app. The user need not enter them every time. However, the user has the option to change the parameters. A more detail understanding on MQTT is included in previous section.

As mentioned above, this function to connect to the MQTT broker is called from the JavaScript file, mqttplugin.java through the function MqttPlugin.prototype.setMQTT(). This file can be found in the plugin folder.

Next we have the function to subscribe to the topic $EDC/tum/B8:27:EB:A6:A9:8A/SOCKET-V1/data. In the java file, the function private String subscribe() facilitates this. The function also handles the operations to be carried out once the message is received at this topic. The subscribe function is also called from the JavaScript file mqttplugin.js through the function MqttPlugin.prototype.subscribe. The same function also handles the operations to be carried out when the data is received. Also note that messages for WiFi are encoded in the Google Protocol Buffers. This is also taken care by this plugin. The decoded information is sent to the JavaScript function. The following code snippet from the mqttplugin.js explains the same.

This function as you can see, also takes care of parsing the message. As in the case of Bluetooth, the message is in the following form: "ForceX=100, ForceY=200, ForceZ=520"

Once the integers values have been extracted, the thresholdAlert function is called followed by the Graph Generation. The implementation for both the threshold and the graph is exactly the same as that of the Bluetooth. The following is a screenshot for the WiFi Screen.

4.3 OPC-UA

The main tasks for this page was to change the state of the led located remotely. This can be extended to any device. (Say a machine in a factory). Another important task for the page is to get the current state of the device. So when the led is switched on, the mobile client also gets the status and a virtual bulb displayed in this page is also turned on and vice versa.

The app subscribe to the topic “tum/led/status”. The Controller will then publish the state of the led on this topic. So as soon as there is a change of state of the led, the controller will publish the state with a payload “on” or “off”. Based on this message, the bulb on the app will change its state.

Similarly, to change the state of the led located remotely, there are two buttons provided on the page. Both these buttons publish on the topic “tum/led”. Based on the Button clicked, “on” or “off” are sent as payload. The controller on the other hand subscribe to the topic “tum/led”. On receiving the “on”, the controller changes the state of the led. The circuit for the led is made on the breadboard, using the GPIO pins. More details on the hardware simulation for the same can be found in Device Simulation section.

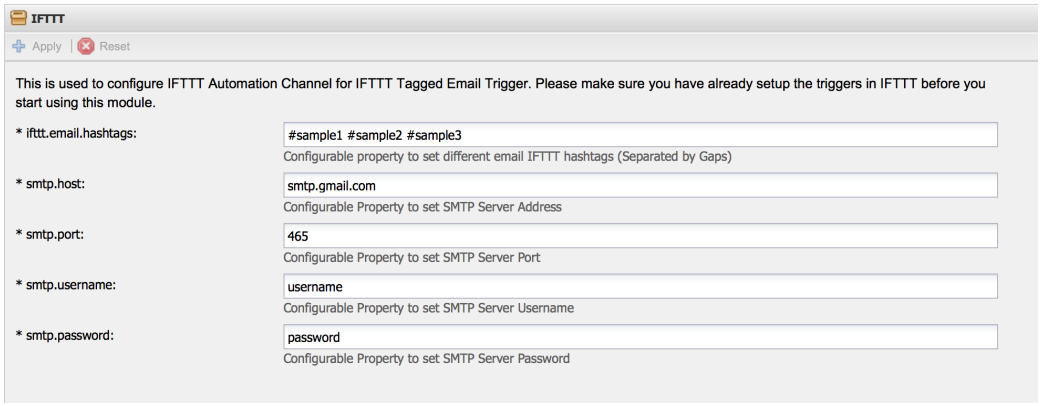
5. Miscellaneous Features in Gateway

5.1 IFTTT

Describing IFTTT to someone unfamiliar with the service can be a challenge. The explanations are often too vague ("IFTTT helps people make simple connections between anything on the Internet") or so complicated they go over people's heads. It's always easier to start with examples. The web service leverages the Internet to automate tasks like saving your Instagram photos to Flickr, sending a text message when rain is in the forecast, turning on your lights from Facebook, and more.

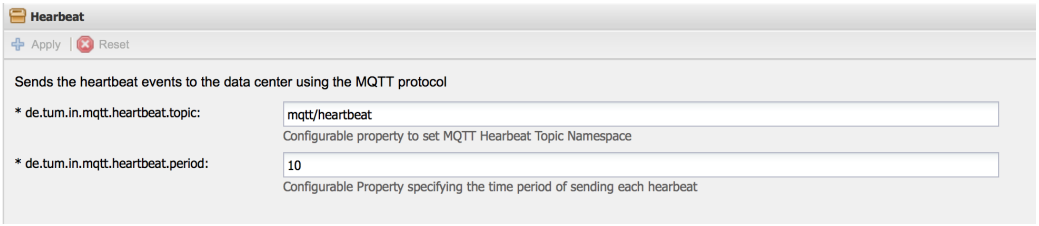
A testament to the Internet of Things, a buzz phrase that refers to networked smart devices saturating our lives—IFTTT carries out actions on the web when channels are triggered. These actions are based on its namesake recipe: If this, then that. It's a simple idea with endless possibilities.

In our situation, we have crated the OSGi micro-service so that people can make use of it in other plugins. The configuration could be done using the Gateway's web UI. Please have a look at the image below. Here you configure the IFTTT registered email ID and the smtp server info to use through which you trigger specific events configured by you in IFTTT using email hashtags.



5.2 Heartbeat

This feature ensures the user about the Gateway to be alive. This feature continuously broadcasts heartbeat in a times fashion. The heartbeat time is also configurable from the Gateway UI. Please have a glance on the screenshot as shown below.



5.3 Real-time Data Cache

The IoT Gateway incorporates a feature ensuring user that the real-time data will never be lost. As soon as the Gateway consumes the data from the field devices, it stores the data locally in its cache for few hours and as soon as it finds good internet connection, it dumps the data to the logging component and removes the cache locally.

6. Device Simulation

6.1 Bluetooth

The Simulation Bluetooth device works on Bluetooth BlueZ stack and it is developed using Python API. The following plugin is responsible to communicate with the Bluetooth Client device for real-time data collection.

The simulation device once connected to the Client Device (i.e IoT Gateway), continuously broadcasts real-time data in the format of forceX=some-no, forceY=some-no, forceZ=some-no, torqueX=some-no, torqueY=some-no, torqueZ=some-no, time=timestamp, type=bluetooth.

The IoT Gateway collects the data from the Simulation Bluetooth device and sends it to the mobile clients for real-time data access and also to the loggers subscription topic so that it the data can also be effectively dumped for further future use.

Responsible Plugin:

1. de.tum.in.bluetooth.milling-machine

6.2 WiFi

The Simulation WiFi device also works on Socket NIO and it is developed using Java NIO API. The following plugin is responsible to communicate with the WiFi Socket Client device for real-time data collection.

The simulation device once connected to the Client Device (i.e IoT Gateway), continuously broadcasts real-time data in the format of forceX=some-no, forceY=some-no, forceZ=some-no, torqueX=some-no, torqueY=some-no, torqueZ=some-no, time=timestamp, type=wifi.

The IoT Gateway collects the data from the Simulation WiFi device and sends it to the mobile clients for real-time data access and also to the loggers subscription topic so that it the data can also be effectively dumped for further future use.

Responsible Plugin:

1. de.tum.in.socket.server

6.3 OPC-UA

The Simulation OPC-UA device works on UA-stack by digitalpetri and it is developed using Java OPC-UA Stack Core API. The following plugin is responsible to communicate with the OPC-UA Client device for real-time data collection.

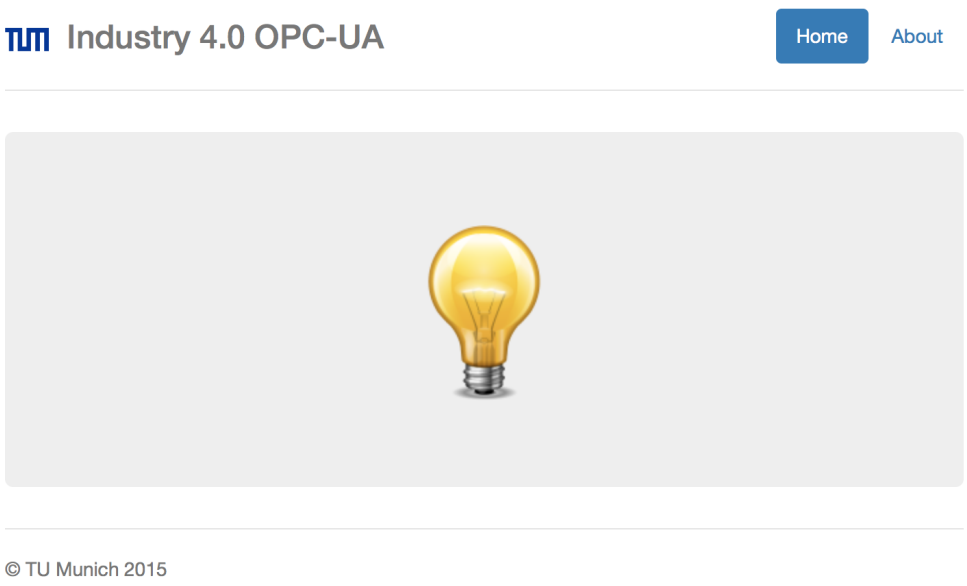
The simulation device is not used for real-time data collection as OPC-UA is not a real-time protocol. OPC-UA as described earlier in its respective section, is mainly used as a device management protocol. That's why we have incorporated LED with GPIO so that the OPC-UA Client (IoT gateway) sends specific switch on/off command to the LED as soon as it receives instructions from the mobile client using OPC-UA server.

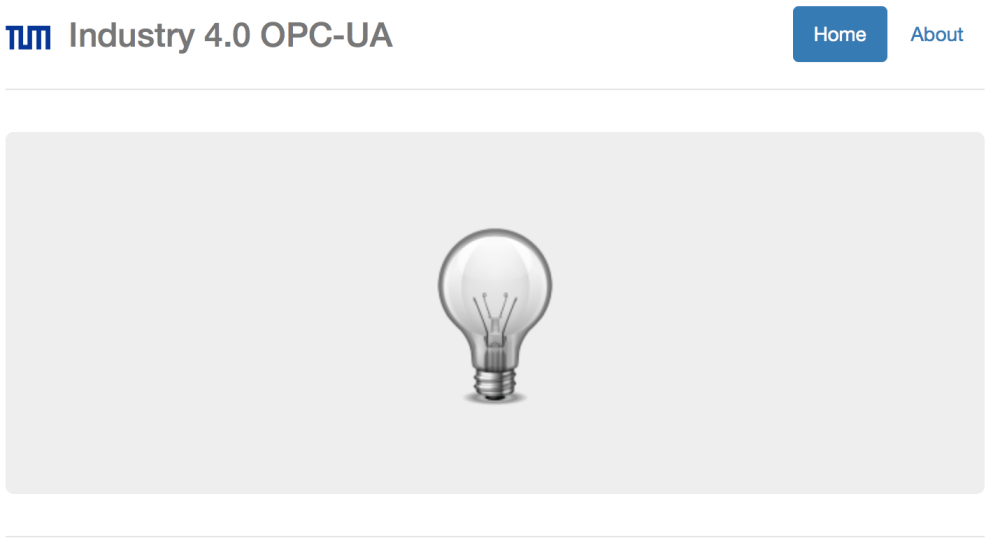
The IoT Gateway receives the proper operational instruction from the mobile client and the gateway then operates directly on the LED's GPIO using OPC-UA.

Responsible Plugin:

1. de.tum.in.led.controller.application

The current application is built on OSGi enRoute. Have a look on the following screenshot.




Industry 4.0 OPC-UA
Home
About
© TU Munich 2015

7. Diving into the Data

7.1 Realtime Data Visualization

We have incorporated one utilitarian tool to visualize the realtime data collection in a Web UI for further analysis on the data. Please have a look at the following to get an idea on how it looks.

Industry 4.0 DWH

Bluetooth Wi-Fi

Realtime Bluetooth Data

Force X	Force Y	Force Z	Torque X	Torque Y	Torque Z
936	714	582	810	600	786
266	985	747	769	120	263
5	34	121	556	507	165
668	132	629	176	83	170
605	305	318	131	116	103
541	585	698	779	641	303
933	875	317	975	340	721
239	136	522	772	745	303
445	202	973	972	138	511
230	319	645	245	629	947
69	132	958	544	663	388
74	368	447	774	391	36
329	2	430	985	662	733
883	58	339	567	963	275

8. How to Run

8.1 Gateway

Follow the below-mentioned steps to get the system up and running:

1. Connect to the network (either using WiFi or Wired)
2. Wait for device to obtain IP Address in the same network
3. Open any browser in your own laptop which is connected to the same network
4. Access the IP Address obtained by this Gateway in your laptop's browser
5. It will prompt you with Login. Use **admin** as username and password both.
6. Now you can access the Gateway and configure it according to your own needs.
7. Currently for demo purposes, we have configured everything to defaults
8. Now it's time to configure the simulations devices
9. Please follow the instructions as mentioned below to start the simulation devices
10. After you finish starting up the devices, you can resume from here

8.2 Simulation Devices

Follow the below-mentioned steps:

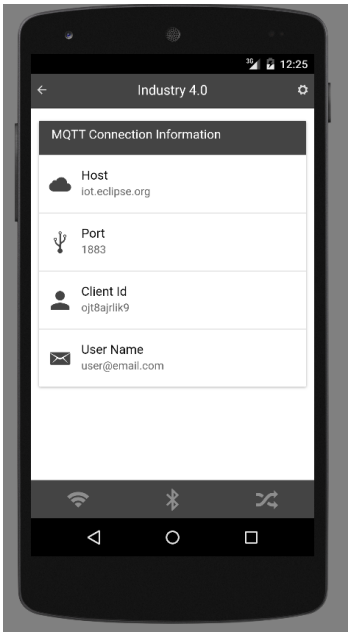
1. Connect to the same network (either using WiFi or Wired)
2. Wait for devices to obtain IP Addresses in the same network
3. You can ssh into the devices using its obtained IP Addresses
4. Please make sure to use **pi** as username and **iotiwbiot** as password
5. If you are logged into WiFi Simulation Device, please follow the respective commands sequentially
6. cd /home/pi/TUM
7. java -jar de.tum.in.socket.server-1.0-SNAPSHOT-jar-with-dependencies.jar 172.20.10.3 9999
8. 172.20.10.3 - the IP Address obtained by the device during our demo. It will change in your setup.
9. Make sure to change the IP Address in the aforementioned command and in the Gateway Web UI
10. The socket server is also configured in the same device.
11. You can start Bluetooth using the following command
12. cd /home/pi/TUM
13. python bt.py machine 00:1A:7D:DA:71:15
14. If you are logged into OPC-UA simulation device, you can start executing the following commands to start.
15. cd /home/pi/Desktop
16. java -jar IDP_TUM_led.jar
17. This might need 2-3 minutes to start
18. You can access the Web Interface at http://[IP-ADDRESS OF OPC-UA DEVICE]:8080

8.3 Mobile Client

Follow the below-mentioned steps:

1. Make sure to have an internet connection in your mobile
2. Start the Mobile Application

3. It will try to get connected to the Raspberry Pi. If it is unable to connect, don't panic. After 30 seconds, it will ask you to check your MQTT configurations. Enter the following configurations.

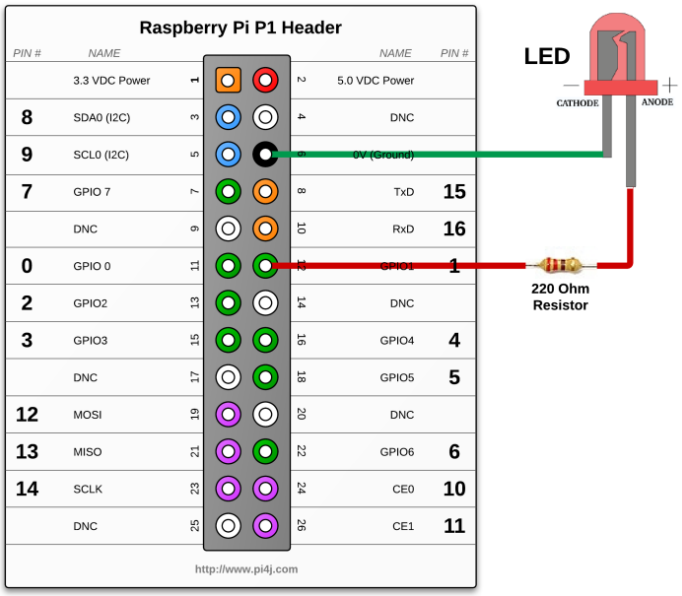


4. After entering the MQTT configurations, wait for the App to get connected, you will see a green signal. Click the signal. You are ready to go!

5. Now you have the options to see the real time data for Bluetooth and WiFi. You also have the option to control the OPC UA devices.
6. Click on the Bluetooth button at the bottom of our screen. It will open the Bluetooth Page. In about 5-6 seconds, the app will receive the Bluetooth data from the Bluetooth simulated device and the graphs will be generated. You can also change the graphs for any Axis. Just click on the respective Force. It is really easy! Also note, that you will also see notifications next to the Force values if the Forces exceed the threshold values. To configure the threshold values, refer to step 9.

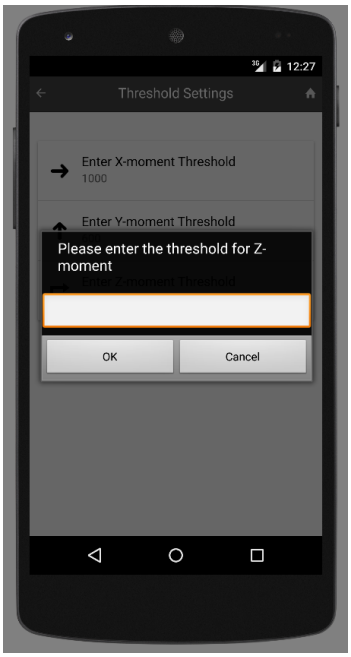
7. Follow the same process for WiFi that you followed for Bluetooth.

8. For the OPCUA, click the on/off buttons. If your circuit is configured correctly, you should be able to control that led. If it does not work, check your circuit as follows.

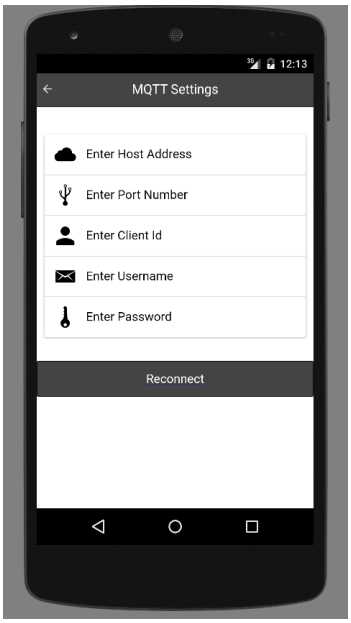


9. If everything is configured as aforementioned, you will get to see the following screen.

10. You can change the threshold values of the forces at any time.



11. Lastly, you can also change the MQTT settings.



9. Contact Persons

ID	Component	Person	Email-ID
1	IoT Gateway	Amit Kumar Mondal	admin@amitinside.com
2	Real-time Data Visualization Application	Amit Kumar Mondal	admin@amitinside.com
3	Device Simulation	Amit Kumar Mondal	admin@amitinside.com
4	Mobile Client (Backend)	Ankur Bhatia	bhatia.ankur8@gmail.com
5	Mobile Client (Frontend)	Pulak Chakraborty	pulakchakraborty.contact@gmail.com
6	Data Analysis	Rupam Bhattacharya	rupam.speaks@gmail.com

Bibliographie

[1] http://www.eclipse.org/kura/

[2] https://www.osgi.org

[3] http://mqtt.org

[4] https://cordova.apache.org

[5] https://opcfoundation.org

[6] http://canvasjs.com

[7] https://www.mongodb.com

[8] http://enroute.osgi.org

[9] http://www.splunk.com

[10] http://pi4j.com