SMART WATER MANAGEMENT SYSTEM

GUIDED BY
R. SIVASAKTHI

S. SNEHA
R. INDHIRA
M. DEVA
V. NIRESHKUMAR

DESCRIBE THE PROJECTS OBJECTIVES,
IOT SENSOR SETUP,
MOBILE APP DEVELOPMENT,
RASPBERRY PI INTEGRATION AND
CODE IMPLEMENTATION

EXPLAIN HOW THE REAL-TIME WATER
CONSUMPTION MONITORING SYSTEM
CAN PROMOTE WATER CONSERVATION
AND SUSTAINABLE PRACTICES

MAINTMASTER IOT SENSORS ARE UNIQUELY DIFFERENT

Plug and play addon to MaintMaster CMMS

MaintMaster IoT Sensors are designed to detect deviations long before breakdowns occur. Sensors are not a new technology. But to connect them to a maintenance system such as MaintMaster is where the real value is generated.

Data from the sensors will be fed directly into MaintMaster. There you can follow trends and set trigger alerts for deviations. Once a deviation is detected, a work order will be created on the right machine and sent to the right competence.

IOT SENSOR USE CASES

Discover all the ways you can use MaintMaster's IoT Sensors to improve your condition-based maintenance.

Hard-to-reach Places

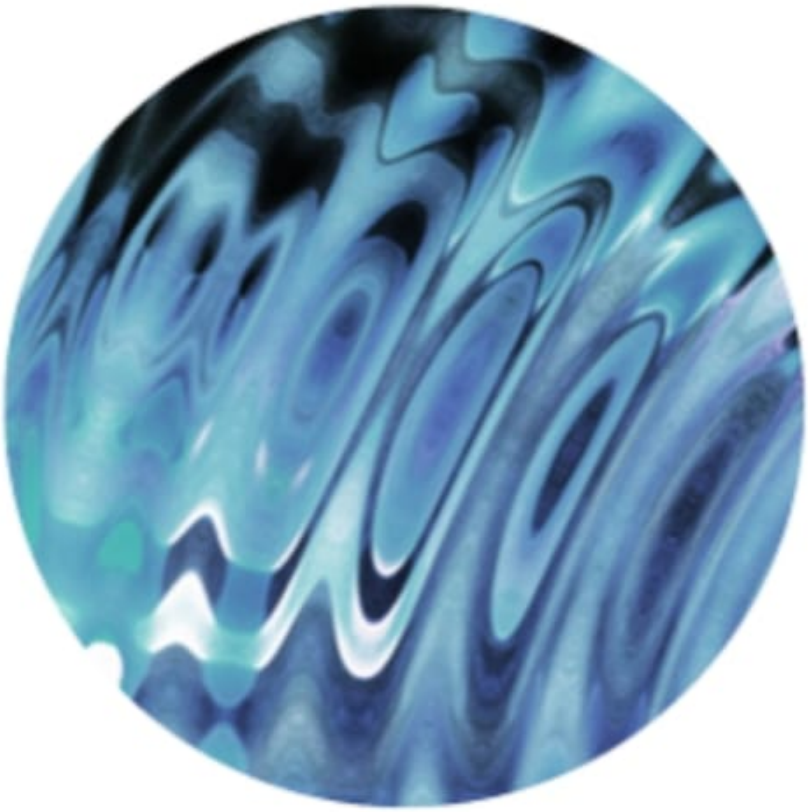
Scaling Monitoring

Vibrations

OUR IOT SENSORS WILL DETECT

Out of the box, our IoT Sensors will measure four things: Air pressure, temperature, vibrations and humidity. And, of course, battery status.
All are visibly in your familiar interface of MaintMaster.

Air Pressure



Vibrations

Humidity

Easy to set up

Our IoT sensors are designed in a way that they can be set up in a very short time. They are powered with batteries and communicate wirelessly. This makes it easy, fast and cost-effective to set up sensors in large amounts. Plug and play

Integrates with your other systems

Our sensors feed data directly into MaintMaster. This means you configure all the variables and actions directly from a familiar user interface – From MaintMaster. There is no need for consultants and programmers, if you are familiar with MaintMaster, you will be able to integrate sensor data!

Useful data

Collecting data is easy, but drawing useful conclusions for a more reliable production is the tricky part. The MaintMaster CMMS software is designed to do just that. MaintMaster automatically creates work orders and other maintenance activities based on sensor data as well as all the graphs and diagrams that you need to present and monitor your activities.

Simple triggers

The aim of our sensors is to create alerts and maintenance jobs based on certain limiting values. We have taken the time to predefine a broad set of limiting value suggestions and detections of changing trends that can be configured inside of MaintMaster. Setting up the triggers requires no programming skills. You can do it on your own.

Affordable sensors

Our Sensors are priced at a point that makes it possible for you to monitor all your equipment. You will not have to make a choice between critical and non-critical machines. Just measure everything to ensure more reliable production.

WHEN SHOULD I INSTALL IOT SENSORS?

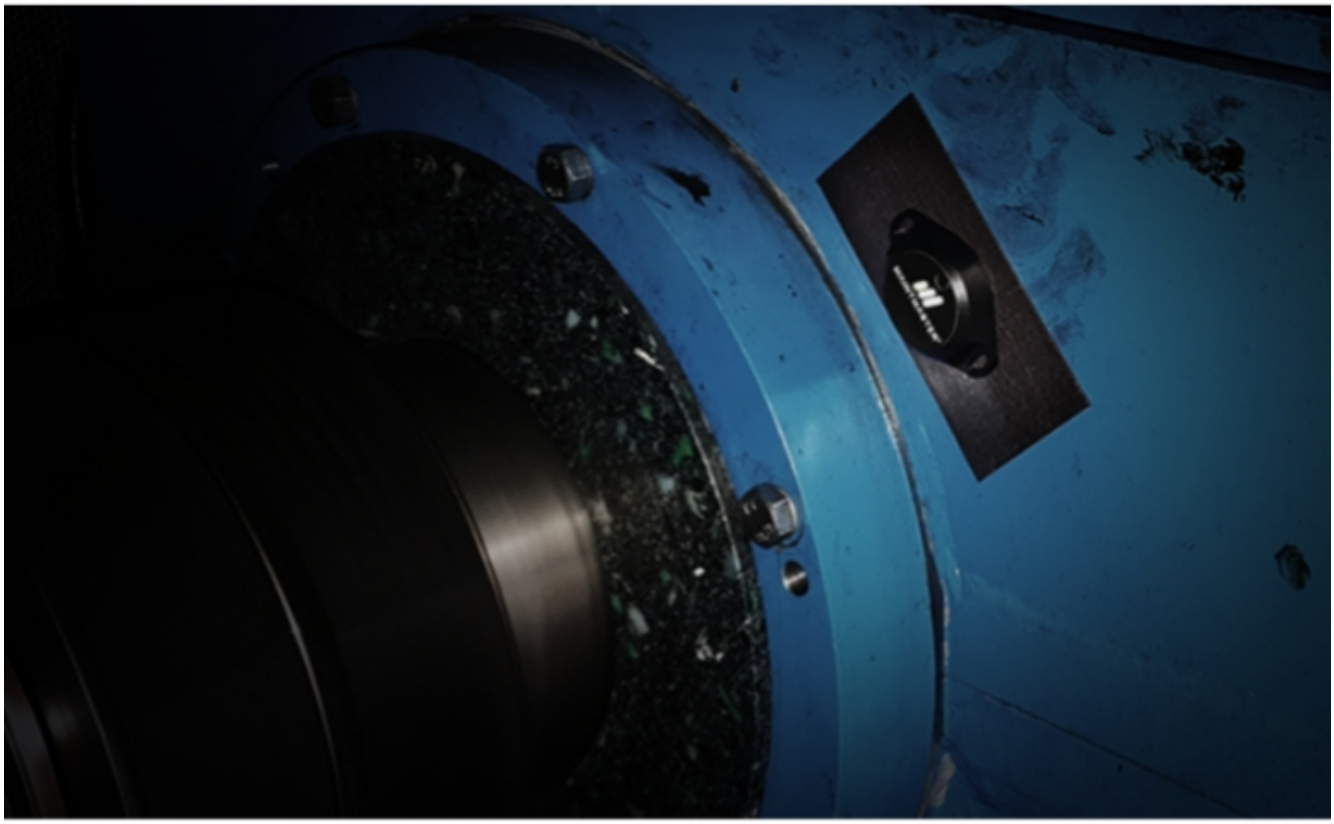
Our IoT sensors have many applications. You can use them to measure vibrations, air pressure, temperature and humidity. That means that there are many machines that can be monitored with MaintMaster IoT sensors.
Getting the data from the sensors and creating tasks from sensor data offers can create a huge benefit to your maintenance department. The most significant value, however, is created in one of the following scenarios.

When inspection requires the production to halt

Some inspections are only possible during a production stop but in many cases, some crucial components of a machine can be monitored with sensors while the production is at full capacity.

When deviations are invisible to human senses

Some deviations can only be observed with the right equipment. Take vibrations for instance: Our vibration sensors can provide reliable data and reliable comparisons over a long period of time.



Some objects are located in far places that are hard to reach. If the procedure to reach them requires dismantling of objects or climbing into high places our sensors can save extensive amounts of work.

When the inspection is a health or safety risk

Some objects are located in places where extensive health and safety equipment is necessary for safe access. Installing sensors instead of having your maintenance personnel inspect dangerous objects has a promising potential to increase workplace safety.

USE CASES

Perfect food management with steady temperatures

Avoid spoilt food or expensive machine breakdown with accurate, real-time temperature monitoring of all your critical equipment. Get automatic notifications and work orders assigned to the right expert when a deviation is detected.

HOT OR COLD? PERFECT.

Maintaining temperatures is vital for many manufacturers handling sensitive foods that need unbroken cold chains from production to distribution. Take the temperature with IoT sensors before deviations indicate incoming breakdowns. And monitor temperature and more data even in extreme temperatures. Get real-time insights and see historical trends to make temperature control easier – with automatic work order creation in your **Maintenance Software**.

READY TO TAKE THE TEMPERATURE?

READY TO TAKE THE TEMPERATURE?

Maintain safe food storage temperatures

Set your desired temperature range and receive instant alerts if temperatures deviate, helping you to prevent potential food spoilage and contamination before it becomes a problem.

Food temperature monitoring from everywhere

Say goodbye to manual inspections and have real-time temperature information from anywhere, at any time, ensuring the freshness and safety of your food.

Let sensors create work orders

Say goodbye to manual work order creation. When a sensor detects a

DISCOVER SENSORS THAT LOVE A CHALLENGE.

Most premises have equipment in isolated or even dangerous places. Plug-and-play sensors allow you to monitor your equipment without risking the safety of your technicians. Rip the tape, place the sensor once and monitor all your hard-to-reach equipment.

Forget dismantling machines or looking for ladders. Get real-time insights and see historical trends to make proactive maintenance easier – with automatic work order creation when the sensors find deviations. Inspect with a click in **MaintMaster CMMS**.

READY TO CUT YOUR NUMBER OF RISKY INSPECTIONS?

Improve your safety

MaintMasters IoT sensors can be placed in hazardous or hard-to-reach locations, enabling remote monitoring and control to reduce risk to human workers.

Increase your efficiency

Our IoT sensors can provide real-time data on conditions in hard-to-reach places, enabling more informed decision-making and optimised operations.

Be better at maintenance

Our IoT sensors can provide early warning signs of potential problems, enabling preventive maintenance to be carried out before failure occurs, reducing downtime and maintenance costs.

WHAT IS IOT APP DEVELOPMENT

- **IoT app development** is the process of building **IoT apps** or software which combine sensor data with machine learning technologies and predictive analytics to create smart and proactive user experiences.

IOT PLATFORM

- An **IoT platform** is a multi-layer technology that enables straightforward provisioning, management, and automation of connected devices within the Internet of Things universe. ... Thus, an **IoT platform** can be wearing different hats depending on how you look at it.

ANDROID ARCHITECTURE

- The software stack is split into four layers
- The application layer
- The application frame work
- The libraries and runtime
- The kernel

HARDWARE

- SOM architecture
- Base board + certified modules
- Custom hardware through pins/connectors/s
- System-on-a-chip Sip
- System-in-package (stacked SOMs)
- Easy certification (CE - EU declaration of conf

IOT - Android Things

IOT COMPONENTS

IOT CODING

- **Android** software development is the process by which applications are created for devices running the **Android** operating system. Google states that "**Android** apps can be written using Kotlin, Java, and C++ languages" using the **Android** software development kit (SDK), while using other languages is also possible.

PROJECT- BUILD IOT APP USING ANDROID THINGS

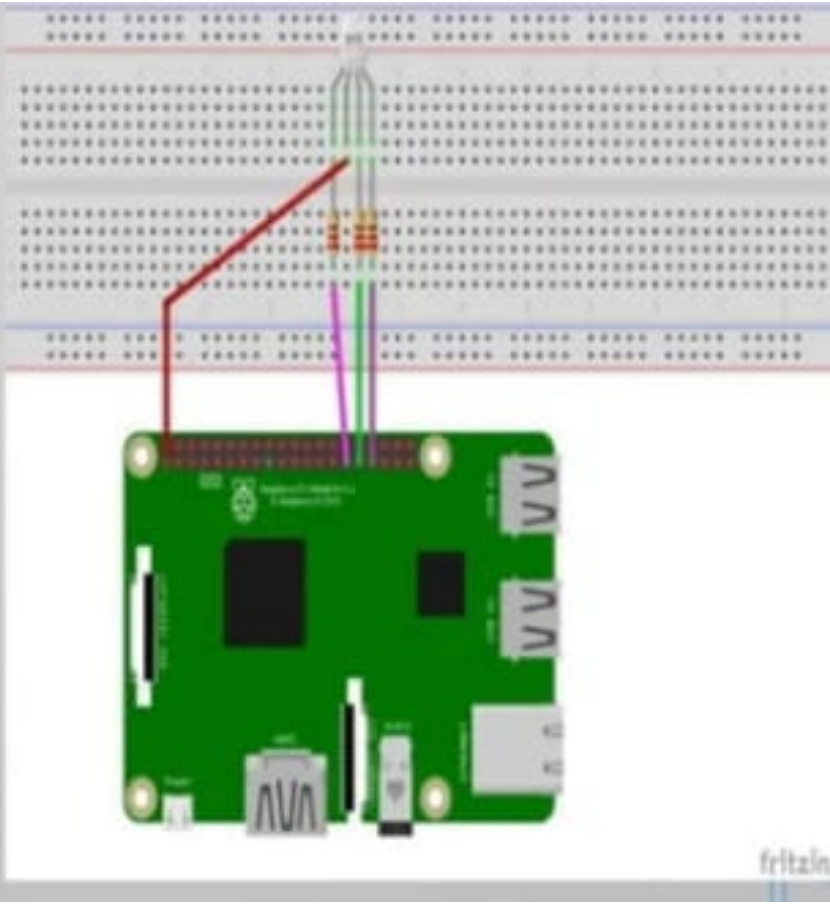
- The goals of this Project are to:
  - Build a simple RGB LED controller.
  - Build an Android IoT app that uses a UI developed using the Android API.

We will use a Raspberry Pi 3 as our IoT prototyping board for this experiment, but you can use any other development board, so long as it's compatible with Android Things.

STEP-I GETTING STARTED

- Usually, an IoT project has two sides: an electric/electronic side and a software side.
- To keep things simple so that we can focus on the Android IoT app, this IoT application controls a simple RGB LED (common anode).

THIS RGB LED IS CONNECTED TO A RASPBERRY PI USING A 220Ω RESISTOR, ONE FOR EACH COLOUR. THE SCHEMATIC DIAGRAM IS SHOWN HERE:



STEP II- ANDROID THING PERIPHERMANAGERSERVICE

- To handle the communication to the RGB LED, we use *GPIO pins*. GPIO pins use a programmable interface to read the device status or to set the output value (High or Low).
- Using Raspberry Pi GPIO pins, we turn on or off the three colour components (red, green, and blue).
- The Android Things SDK provides a service called PeripheralManagerService to abstract the GPIO communication interface.
- We have to use it every time we want to read or write data.

STEP III- ANDROID IOT APP UI

- Another interesting feature provided by Android Things is the UI interface.
- We can develop a UI interface for our Android IoT app in the same way we develop any other Android UI.
- Like with an Android app, the UI is in the XML format.

APPLICATION FRAME WORK

FUTURE SCOPE

- IoT or internet of thing these days we quite often listen to this word.
- The words such as the cyber-attack, hacking we can hear these too jointly with IoT.
- Despite, of the fear of piracy and cyberattack the internet of things and IoT applications will grow much more faster in the coming years.
- Ultimately we can say the IoT future scope or the future of IoT is very bright.

shut down of different devices over time (e.g. mobile nodes, IP cameras, etc.), depending on the movement and trajectory of the tracked object.

In such scenarios, solutions are needed that are based on the usage of *communication middleware* to provide *flexible interactions* while preserving certain ranges of *temporal guarantees* that suit the requirements of the specific application domain. Nevertheless, using distribution middleware has been neglected in areas such as critical systems, especially for those parts of higher criticality levels. This is due to the fact that, traditionally, middleware employs a set of libraries that make use of fairly sophisticated programming techniques rather oriented to provide an easy-to-use programming environment, of high development productivity, with reliable, fast, and efficient communication for mainstream (i.e., general purpose) systems. It mostly focuses on providing a friendly and productive environment than to providing strict temporal guarantees.

Fig. 1. Overall cyber-physical systems with interaction among powerful servers and autonomous low power embedded computers

Providing a flexible infrastructure that, at the same time, preserves timely operation is a complex problem especially in the presence of autonomous and mobile nodes. The target domain is exemplified in figure 1 that shows a large system distributed partly in a factory floor and partly in the cloud. The factory floor part contains a number of subsystems being one of them an autonomous surveillance system made of a number of autonomous mobile devices, equiped with different sensors. These nodes capture video data about their assigned area and transmit them to a server that is in charge of a first processing of the video and its further transmission to the cloud.

Over the last decade, a number of contributions on middleware design have appeared to provide enhanced functionality to distributed systems, but they focus mainly on enlarging the functional dimension of these systems. Also, a number of improvements to the networking protocols have appeared for augmenting bandwith and improving reliability. To the best of our knowledge, there is no contribution that focuses on studying the interaction between heterogeneous nodes in a time-sensitive environment; nor on exploring the possibilities of using de facto middleware standards such as DDS (Data Distribution Service for real-time systems) for communication of nodes with heterogeneous computation

power; this comprises servers (used for heavy processing and data transmission) and embedded computers (used for environment monitoring such as video capture, and readings of valuable data as temperature, humidity, movement, etc.).

This paper analyzes the temporal behavior of the DDS-based interaction between high computation power nodes and ARM-based embedded computers; precisely it focuses on Raspberry Pi devices as the embedded computers, given their high interest as they have gained large market share in both industry and academic worlds. This paper presents a flexible library to communication Raspberry Pi and servers using different underlying communication software (e.g., bare UDP and IP sockets; or data distribution middleware to exploit their quality of service parameters for timeliness and reliability). Our target system integrates heterogeneous nodes (ranging from individual sensors and/or embedded nodes that can be static or autonomously moving in an e.g. surveillance area) and base servers (that interact with the sensors and with other servers).

The paper is structured as follows. Section 2 describes background containing selected related work. Section 3 provides baseline information on the middleware technologies. Section 4 proposes an architecture for integrating Raspberry Pi embedded computers with servers (virtualized and non-virtualized) using DDS as communication backbone, and presents a flexible communication library. Section 5 provides experimental results of the usage of the library in heterogeneous conditions. Section 6 draws the conclusions.

2 Background

Embedded computers are used in a large number of application domains, as they facilitate the development of lower cost solutions that are, in turn, more energy efficient as compared to the usage of high end processors. The computation power of the processors that they use (e.g., ARM family processors) is considerable, supporting the use of comodity operating systems (such as Linux) and user level libraries (such as communication middleware) that aid the development of powerful applications. Middleware employs a set of libraries that make use of fairly sophisticated programming techniques oriented to provide an easy-to-use programming environment with reliable, fast, and efficient communication for general purpose systems. The main focus in on programmability and not on offering strict temporal guarantees. Then, technologies such as the following have been discarded for the most time-sensitive parts of critical systems: Corba [24], Java RMI [29], web services [22], Ice [30], JMS [4], REST, AMQP [20], RabbitMQ [25], Storm [2], River [1], or JBoss [18], among others. DDS [23], the most popular publish/subscribe data-centric standard, offers *quality of service* (QoS) *policies* that allow the designer to fine tune the communication among the endpoints (publishers and subscribers), but it does not guarantee any upper bound on the temporal behavior of the interactions and/or communication times.

The appearance of the cyber-physical systems paradigm envisions large scale systems composed by large numbers of heterogeneous subsystems and nodes across heterogeneous networks. This idea meets the Internet of Things [7]; where

protocols for integration of devices need be designed [8]; and where security and reliability must be addressed [6]. Such complex scenarios require more productive design and development means where communication middleware can play a major role. Middleware for cyber-physical systems will have to meet the requirements exposed in [10] including model-based design, on-line reconfiguration and verification functionality to support system evolution at the pace dictated by changing environment conditions; and performace analysis such as [17] need be done for different domains. Some experiments for on-line reconfigurations using verification have been presented in [3, 12]. In this area, a number of contributions to the design and implementation of real-time middleware have been proposed, but the great majority only provide partial solutions to very specific and limited problems of communication predictability. For instance, [11], a real-time middleware for distributed real-time applications, supports dynamic service-based reconfiguration; [14] presents a dessign of a real-time middleware that fine tunes specific internal parameters (such as thread pool size); [15] presents a distributed architecture based on a central manager that supports a dynamic number of clients connected to specific servers through the remote management of its thread pool; and [13] provides a middleware that is aware of the multicore processor architecture to prioritize selected client requests, improving service times of priority client requests.

Embedded computers are a key enabler for the realization of the cyber-physical systems vision. But these require to comply with some level of temporal guarantees. To the best of our knowledge, there is no contribution on the analysis of performance experienced by embedded computers such as Rasperry Pi based systems for building autonomous systems capable of communicating with servers (that can be virtualized providing a mixed criticality platform) through flexible standard-based paradigms such as the Data Distribution Service.

3 Overview of technologies

Large systems such as those involving IoT and cyber-physical deployments present common characteristics although they have different requirements for temporal guarantees [9]. The following are some of their common characteristics:

- device heterogeneity in what concerns hardware and software characteristics (including the heterogeneous network links);
- devices with heterogeneous resources: some nodes can be sensors with limited computation power and autonomy; embedded computers with moderate but sufficient capacity; and servers with high end hardware and full fledged software;
- dynamic structure and interactions: the set of participant nodes, environment conditions, and requirements may vary over time;
- large scale: nodes, subsystems, and entities can be progressively integrated into larger subsystems;
- event handling complexity: a large number of events can be produced, of different types, and of varying nature (carrying their own information);

– intelligence and distributed management: autonomous decisions are taken considering the overall system conditions and those of its constituent parts.

Execution platforms. The heterogeneous nature of devices impacts the overall operation timeliness. On the high end, there is the presence of powerful many core servers with high performance in data processing and communication. On a lower end, there are embedded computers capable of playing the role of autonomous nodes with low power consumption and considerable computation power. These are typically connected to sensors that are at the lowest end of computation capacity given their restrictions on continued power supply and battery. Raspberry Pi is among the most popular choices in a number of domains. It is a powerful and low-cost platform with good hardware expansion capabilities such as different ports, a GPIO –or General Purpose Input/Output–, pins, etc.) and a range of connectivity possibilities such as Ethernet or WiFi. One of the advantages of Raspberry Pi is low cost that has its origins on its initial educational focus. It has been demonstrated on a number of prototype applications such as health services [19], biometrics [27], control [26], or video systems [28], among others.

Middleware connectivity problem space. The problem of connecting distributed and remote systems (devices, sensors, components, micro controllers, servers, desktops, etc.) is handled with communication middleware for a number of domains and, especially, for those that fall in the category of best effort. Following, some technologies are described with respect to their interaction models, resource efficiency, and level of fine tuning of their quality of service parameters: *Java Messaging Service (JMS)* supports publish-subscribe messaging among multiple nodes. It has two application level routing options: point to point and publish-subscribe, typically over a centralized implementation. It is restricted to Java language and part of the Java Platform Enterprise Edition (Java EE) [5]. *MQTT* is a wire protocol that is message-centric allowing to transfer telemetry-style data provided as messages for application domains requiring low latency. *REST* is an application programming interface (API) for web applications that operate in client-server mode, having a simple interface with a reduced number of operations over resources. A resource is any coherent and meaningful addressable concept that is represented by a document that captures the state of the resource. *AMPQ* [20] (Advanced Messaging Protocol Queue) is an interoperability protocol emerged in the financial sector to avoid propietary and non-interoperable connectivity. Interoperability is achieved by controling the behavior of the messaging provider and the client, i.e., it is a wire protocol. It provides message-oriented communication and flow control.

Data Distribution Service standard. The Data Distribution Service (DDS) is an OMG standard that provides a publish-subscribe (P/S), i.e., a decoupled and data-centric interaction model among remote components. It was designed for connectivity of large scale systems with heterogeneous devices requiring decoupled interoperation. The goal of DDS is to meet the requirements of real-time distributed applications for which a variety of quality of service (QoS) policies are specified. It is, nevertheless, limited by the actual available implementations and the support from the underlying networking level.

DDS relies on the concept of a *global data space* where entities exchange messages based on their type and content. Such entities are *remote nodes* or *remote processes*, although it is also possible to communicate from within the same local machine. Entities can take two roles; they can be *publishers* or *subscribers* of a given data type. Types are based on the concept of *topics* that are constructions supporting the actual data exchange. Topics are identified by a unique name, a data type and a set of QoS policies; also, they can use *keys* that enable the existance of different instances of a topic so that the receiving entities can differentiate the data source. Applications organize the communicating entities into *domains*. Essentially, a domain defines an application range where communication among related entities (an application) can be established. A domain becomes alive when a *participant* is created. A participant is an entity that owns a set of resources such as memory and transport. If an application has different transport needs, then two participants can be created. A participant may contain the following child entities: *publishers*, *subscribers*, *data writers*, *data readers*, and *topics*. Publishers and subscribers are an abstraction that manage the communication part, whereas the data writers and data readers are the abstractions that actually inject and retrieve the data.

4 Integration architecture

4.1 Target system

The target scenario is presented in figure 2. It is particularized for a smart factory system composed of a number of nodes interacting to perform different functions towards a global goal of monitoring the operation of the factory floor that is divided into different cells. In this scenario, timely monitoring of the operation is performed to obtain useful on-line feedback to later adjust the operation in real-time and maximize the operation benefit.

For achieving such a timely supervision and actuation process, the following actors are present:

- *Autonomous monitoring nodes* based on *embedded computers (EC)* that perform surveillance activities over their corresponding cell collecting monitored data through a number of sensors. ECs collect data from the physical system and area that they monitor. Data are sent to servers for further processing and cross-processing.
- *Base servers (BS)* or cell processing nodes that are computers acting as base stations for collecting the data monitored by ECs. They also perform initial checks on the monitored data. One of the BS, has the role of a gateway node that connects the cell to the outside.
- *Data center (DC)* is a system in the factory for smart data processing. The factory also has a private data center for additional and heavy processing of monitored data; also, it is used for execution of other functions such as ERP (Enterprise Resource Planning). The factory is also connected to an external cloud where specialized (possibly outsourced) processing can be carried out.

Fig. 2. Target system exemplified on a smart factory scenario

Temporal overheads have to be controlled to guarantee the timely operation on the factory processes. For this, an analysis of possible bottlenecks must be performed. One of the most important communication links to check is the interaction between the embedded computers and the base servers: BS-EC link.

The overall operation and communication process of ECs and BSs must be timely in order to perform real-time monitoring of the system status, decision, and actuation. The overall process comprises the following activities:

- raw data collection from the production cells;
- data pre-processing at cell level;
- transmission of raw data and pre-processed data to data center;
- deep processing of data at the data center; and
- transmission of actuation parameters to the cell actors.

As ECs are autonomus, they can move about their allowed surveillance zone (i.e. their cells) to monitor the status of the system. Then, ECs will use wireless or wired connection, depending on their mobility degree. Once ECs collect data, they send it to the BS to perform additional processing and computation to control the actuation over the physical processes.

4.2 Architecture overview: Sotware stack for BS-EC links

The interaction between BS and EC uses middleware as part of a mixed criticality architecture that enables to consolidate BS usage to run applications or functions with different criticality levels in an isolated way. Figure 3 presents the software stack for the involved nodes. The software stack within each node type (BS or EC) follows the lines indicated below:

- *Mixed criticality system* principles. Different applications can coexist without interfering in their execution in the BS, that will be able to simultaneously run a number of functionalities such as the control of the physical process (e.g. a conveyor belt) and the interaction with an embedded computer.
- *Partitioned system*. Partitioning is a technique followed in real-time systems to achieve temporal and spatial isolation across applications over the same physical machine. BS are partitioned systems because they run different

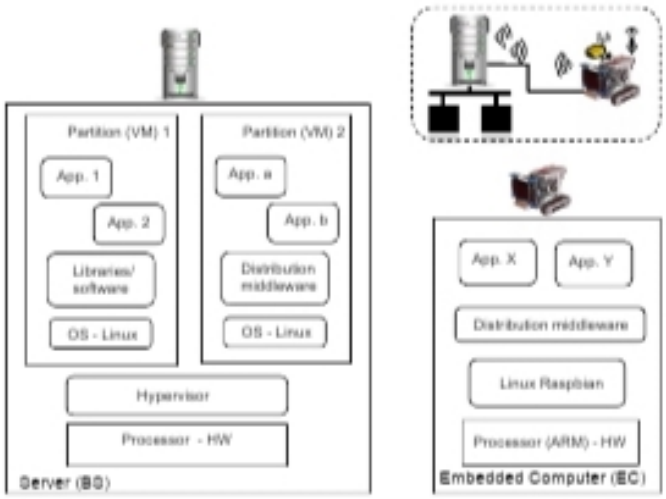


Fig. 3. Software stack for BS-EC link in a mixed criticality environment

partitions or virtual machines (VMs) as part of the software stack. Figure 3) shows the partitions (i.e., virtual machines), the distribution middleware and layers of the BS software stack. It has a hypervisor or a virtual machine monitor that supports ARINC 653 scheduling for ensuring non-interference across the execution of partitions. Partitions have the notion of full execution isolation, whereas the mainstream related term is virtual machines. For this, a BS can run additional functions for performing the intensive data pre-processing in parallel with running the software for communication with ECs, without affecting the interaction time with ECs. A simple model for scheduling partitions using DDS in a distributed context is given in [16]. For each partition, the guest operating system performs the scheduling and management of the execution of the application functions inside that partition.

– *Decoupled distributed software.* The BS runs distribution middleware (precisely, Data Distribution Service) for decoupled communication with ECs. ECs require to send data to BSs and also BSs can send information and/or operation commands to BS. The effect of possible network delays is protected by accounting for the communication times on the execution time assigned to the communicating partitions.

4.3 Programming the communication

A library is provided for supporting the communication among BS and EC. The library provides flexible interaction across the nodes as the underlying communication protocol can be modified, (e.g., basic IP or UDP sockets or Data Distribution Service) by means of a simple interface. The following code provides the main class `BSEC_Flexcom` listing only the two main functions for communication between BS-EC (`bsec_transmit` and `bsec_receive`).

Listing 1.1. Basic communication functions

```
class BSEC_Flexcom {
  public:
    ret_code bsec_transmit(Data_Msg *);
    ret_code bsec_receive(Data_Msg *);
}
```

Listing 1.1 shows an overview of the implementation over a Data Distribution Service communication backbone. Both send and receive functions are sketched.

Listing 1.2. Implementation over data distribution system

```
struct ADataType { // content };
#pragma keylist ADataType id
dds::Topic<ADataType> tsTopic("ATopic");

ret_code bsec_transmit(Data_Msg *){
 dds::Topic<ADataType> tsTopic("ATopic");
 dds::DataWriter<ADataType> dw(tsTopic);
 ATopicType ts = msg_content_fill();
 dw.write(ts);
 return check_status();
}

ret_code receive(Data_Msg *){
 dds::Topic<ADataType> tsTopic("ATopic");
 dds::DataReader<ADataType> dr(tsTopic);
 dds::SampleInfoSeq info;
 ADataSeq data;
 dr.read(data, info);
}
```

Listing 1.3. Setting of a periodic transmission with a reliable writer

```
<protocol>
 <rtps_reliable_writer>
  <disable_positive_acks_min_sample_keep_duration>
   <sec>DURATION_ZERO_SEC</sec>
   <nanosec>100000000</nanosec>
  </disable_positive_acks_min_sample_keep_duration>
 </rtps_reliable_writer>
</protocol>
```

For the case of the implementation over DDS, it is possible to fine tune the communication by setting the appropriate values for the quality of service policies. This will adjust the communication performance as needed. Examples of fine tunning are disabling acknowledgement for sample delivery (see listing 1.3 for setting to $100ms$).

5 Experimental results

An experimental setting has been deployed to analyze the different relevant versions of the BS-EC link. BS hardware is based on Intel i7-4770 processor at 3.4 GHz, 8MB cache, with 8GB DDR3 1600 RAM. BS runs a Linux Ubuntu 15.10 and kernel 4.2. The embedded processor is a Raspberry Pi with an ARM1176JZFS processor (i.e., ARM11 using an ARMv6-architecture core) with floating point unit, running at 700Mhz, with 512 MB. A Netgear CG3300 router L2/L3 is used both for Ethernet (GigaE) and Wireless 802.11n, running a Raspian Whezzy operatinvg system, replacing the kernel to be 3.18.9-rt5-v7.

Experiments show the performance of the communication of Raspberry Pi ECs using DDS Connext 5.2.3 for communication to/from BS nodes. Measurements are performed over sequences of 1000 trials. Firstly, a control group over the bare machine server has been provided; then, the virtualized server has been configured to obtain the communication performance. Experiments also show different load conditions for both the server (PC) and the embedded computer (RPi). BSs are virtualized with Xen. The publisher node has a transmission period of $100\mu s$, for all scenarios.

Figure 4 shows the temporal behavior for the communication between server and embedded node for a scenario where the server is not virtualized (bare server) and it is progressively loaded. Experiments are conducted messages that are, both, smaller and larger than the MTU (maximum transmission unit). For different load conditions, the figure shows the communication cost for both directions:

- PC(P) - RPi (S): from the server to embedded computer; the base station acts as the publisher (P) and the embedded computer as the subscriber (S).
- RPI(P) - PC(S): from the embedded computer (publisher, P) to the base station (subscriber, S).

The maximum transmission time is similar for all load contitions, being the real maximum $104.8\mu s$ for messages larger than MTU size and $5.1ms$ for smaller ones. It is observed that the situation when the base station is the publisher and the embedded node is the subscriber outperforms the opposite situation; this is observed for, both, message sizes larger and smaller than the MTU. Times are overall stable as maximum and minimum cases are consistently around the average for practically all cases. The average times are consistently around the same value for both cases: $100ms$ for messages smaller than MTU and $1.3ms$ for messages larger than the MTU. Variations are minor with respect to the execution over a bare server. Worst cases for messages larger than MTU are consistently around the same value ($4.8ms$), except for the near empty load scenario.

Figure 5 shows the communication time for the link BS-EC for a scenario where the embedded node is progressively loaded and the BS is not virtualized. Times are, overall, slightly less stable than in the previous case, ranging between $1.2ms$ for messages smaller than MTU and $2.4ms$ for the larger messages. Also, publisher BS and subscriber EC yield the best performance.

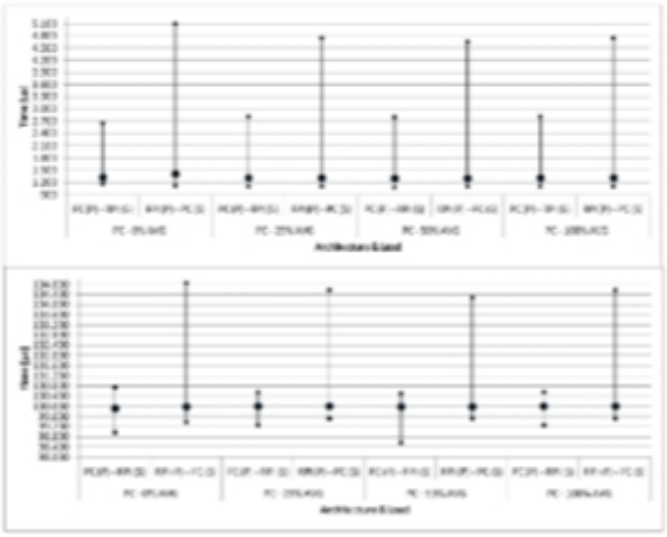


Fig. 4. Communication BS-EC for a progressively loaded bare BS. Top and bottom graphs show behavior for messages larger and smaller than MTU, respectively

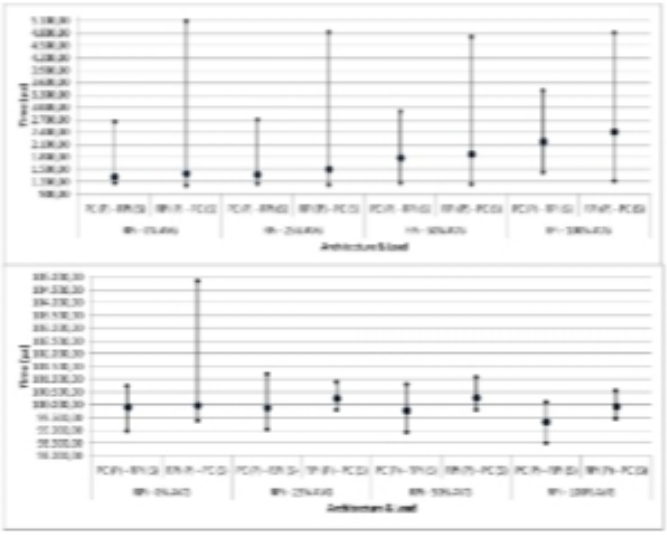


Fig. 5. Communication BS-EC for virtualized BS and progressively loaded EC

Figure 6 shows a virtualized BS that is progressively loaded. The average case ranges from $64.65ms$ to $64.9ms$ for messages larger than MTU; and from $1.23\mu s$ to $1.05\mu s$ for the smaller messages.

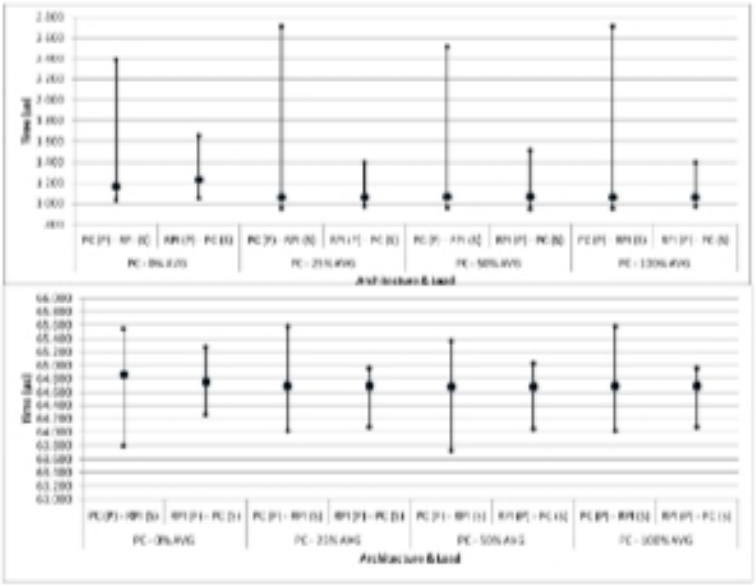


Fig. 6. Communication BS-EC for a virtualized BS that is progressively loaded

Figure 7 shows the temporal behavior for the communication when both nodes are progressively loaded simultaneously. In this case, worst case times increase, although the average times are constantly around $100\mu s$. The average case time experiments greater variation (between $1.1ms$ and $2.2ms$) for the case of smaller message size. For messages larger than MTU, the time increases slightly as compared to the previous case, from $64.25ms$ to $64.9ms$

Table 8 presents a summary of the experiments comparing the above conditions to the scenarios over bare machine and for message sizes larger that the MTU. The *result* column indicates whether the BS is or not virtualized.

Overall, results show that the communication is stable both for bare machine and virtualized settings, for different load conditions and message sizes within the MTU limits and beyond. Worst case values are below 5% larger than the average cases. It is evidenced that Raspeberry Pi communication using DDS shows stable behavior with dispersions that, for the largest case reach $438\mu s$.

6 Conclusions

This paper analyzes the communication timeliness in the context of a distributed monitoring architecture based on autonomous embedded computers and servers

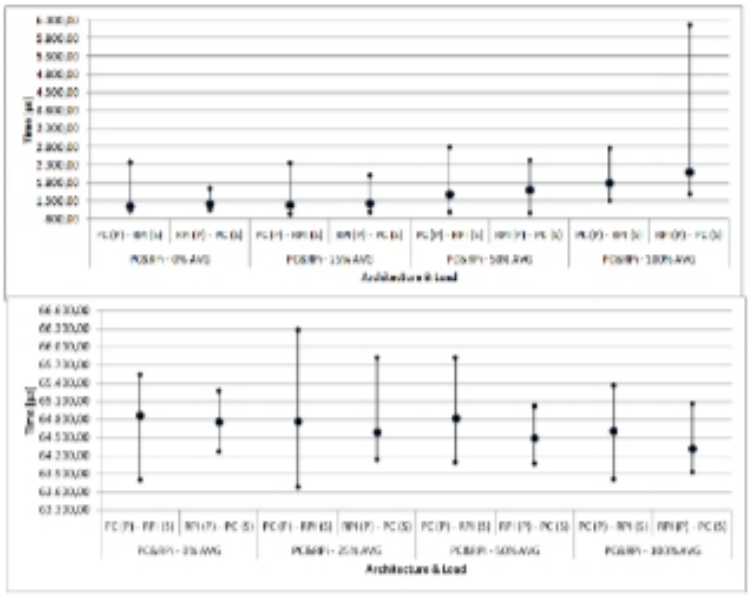


Fig. 7. Communication BS-EC for progressive loads at both nodes

		DATA LENGTH: 66 BYTES			
		Sockets UDP	Result	Middleware DDS	Result
0% LOAD	Virtualized	803,85 µs	Non-Virtualized	1199,01 µs	Virtualized
	Non-Virtualized	736,11 µs		1370,45 µs	
90-100% LOAD	Virtualized	1337,78 µs	Non-Virtualized	1738,2 µs	Virtualized
	Non-Virtualized	699,18 µs		1986,82 µs	
		DATA LENGTH: 2 63000 BYTES			
		Sockets UDP	Result	Middleware DDS	Result
0% LOAD	Virtualized	795,26 µs	Non-Virtualized	64814,39 µs	Virtualized
	Non-Virtualized	742,91 µs		99955,09 µs	
90-100% LOAD	Virtualized	1269,82 µs	Non-Virtualized	64554,63 µs	Virtualized
	Non-Virtualized	715,74 µs		99832,65 µs	

Fig. 8. Summarized analysis of P/S enabled communication for both bare and virtualized settings.

that can support mixed criticality execution. Unlike in traditional cyber-physical systems requiring real-time behavior (that use the level 2 networking protocols to control access to the media and achieve predictable network schedules), our deployment uses a Data Distribution Service communication middleware over

Iot water consumption monitoring

[1]Gollapalli Ashok kumar, [2]G.Sai Poornima [3]Ch.Manisha [4]G.Poojitha
[1]Assistant Professor, [2,3,4]B.Tech Final Year Student
Department of EEE
Narayana Engineering College Gudur, Andhra Pradesh, India.

Abstract: *O*ne of the biggest problems faced today is the challenge of water scarcity caused by increasing water consumption. This consumption can be intimated by measuring the flow of water to every connection in a water supplying network. In this paper, we are focusing on continuous and real time monitoring of water supply in IOT platform. Water supply with continuous monitoring makes a proper distribution so that, we can have a record of available amount of water in tanks, flow rate, abnormality in distribution line. Internet of things is nothing but the network of physical objects embedded with electronics, sensors ,software and network connectivity where monitoring can be done from anywhere .

Keywords **– IOT Platform, NodeMcu, Water flow sensor, Serial monitor**

I. INTRODUCTION

Water is the most precious and valuable resource because it is a basic need for all the human beings but, now a days water supply department is facing many problems in real time operation this is because less amount of water in resources due to less rain fall. With increase in Population, urban residential areas have increased because of this reasons water has become a crucial problem which affects the problem of water distribution, interrupted water supply, water conservation, water consumption and also the water quality so, to overcome water supply related problems and make system efficient there is need of proper monitoring and controlling system. In this paper, we are focusing on overcoming the water scarcity caused by increasing water consumption. This can be done by intimating the water consumption by measuring the flow of water in every connection in a water supplying network. On the basis of the measured value, the usage of water by each unit can be calculated.

2. OVER VIEW OF IOT WATER MONITORING SYSTEM:

Fig:1 Iot water monitoring system

3. HARDWARE REQUIREMENTS:

1.NODEMCU
2.WATER FLOW SENSOR
3.BREADBOARD
4.WIFI CONNECTIVITY
5. JUMPER WIRES

3.1 NODEMCU

NodeMCU is an open source IOT platform. It includes firmware which runs on the ESP8266 Wi-Fi SoC from Espressif Systems, and hardware which is based on the ESP-12 module.

Fig -2 NodeMcu

3.2 WATER FLOW SENSOR

Water flow sensor consists of a plastic valve body, a water rotor, and a hall-effect sensor. When water flows through the rotor, rotor rolls. Its speed changes with different rate of flow. The hall-effect sensor outputs the corresponding pulse signal. This one is suitable to detect flow in water dispenser or coffee machine.

Fig –3 water flow sensor

4. Hard Ware Specifications:

Table -1: specification parameters

S. No	Component	Ratings
1.	NODEMCU	-------
2.	WETER FLOW SENSOR	5 to 18V DC
3.	LED	------

5.CONNECTION:

The hardware connection is done by attaching 3 cables (jumper cables) between the flowmeter and the nodemcu, in order to get the on/off pulse generated by the flow of water and thus count for the litres .

- Connect the red cable output from the flowmeter to the 3.3v pin on the nodemcu.
- The black cable output from the flowmeter to "ground" on the nodemcu.
- Connect the yellow cable output from the flowmeter to the control pin on the nodemcu.
-

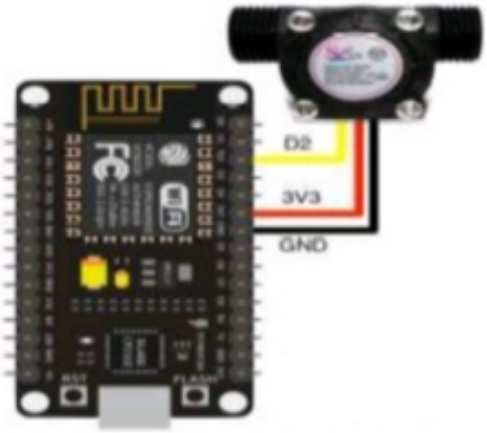


Fig:4 Connection Diagram

6.CODE FOR NODEMCU

```
#include <Arduino.h>
#include <EEPROM.h>
#define USE_SERIAL Serial
#include <ESP8266WiFi.h>
#include <ESP8266HTTPClient.h>
// Variable init
const int buttonPin = D2; // variable for D2 pin
const int ledPin = D7;
char push_data[200]; //string used to send info to the server ThingSpeak
int addr = 0; //endereço eeprom
byte sensorInterrupt = 0; // 0 = digital pin 2
// The hall-effect flow sensor outputs approximately 4.5 pulses per second per
// litre/minute of flow.
float calibrationFactor = 4.5;
volatile byte pulseCount;
float flowRate;
unsigned int flowMilliLitres;
unsigned long totalMilliLitres;
unsigned long oldTime;
//SSID and PASSWORD for the AP (swap the XXXXX for real ssid and password )
const char * ssid = "<NETWORK_NAME>";
const char * password = "<NETWORK_PASSWORD>";
//HTTP client init
HTTPClient http;
void setup() {
Serial.begin(115200); // Start the Serial communication to send messages to the
computer
delay(10);
Serial.println('\n');
startWIFI();
 // Initialization of the variable "buttonPin" as INPUT (D2 pin)
    pinMode(buttonPin, INPUT);
// Two types of blinking
    // 1: Connecting to Wifi
```

```
    // 2: Push data to the cloud
 pinMode(ledPin, OUTPUT);
 pulseCount = 0;
 flowRate = 0.0;
 flowMilliLitres = 0;
 totalMilliLitres = 0;
  oldTime = 0;
 digitalWrite(buttonPin, HIGH);
 attachInterrupt(digitalPinToInterrupt(buttonPin), pulseCounter, RISING);
}
void loop() {
if (WiFi.status() == WL_CONNECTED && (millis() - oldTime) > 1000) // Only process
counters once per second
 {
 // Disable the interrupt while calculating flow rate and sending the value to
 // the host
   detachInterrupt(sensorInterrupt);
 // Because this loop may not complete in exactly 1 second intervals we calculate
  flowRate = ((1000.0 / (millis() - oldTime)) * pulseCount) / calibrationFactor;
  oldTime = millis();
  // convert to millilitres.
  flowMilliLitres = (flowRate / 60) * 1000;
  // Add the millilitres passed in this second to the cumulative total
   totalMilliLitres += flowMilliLitres;
   unsigned int frac;
   // Print the flow rate for this second in litres / minute
   Serial.print("Flow rate: ");
   Serial.print(int(flowRate)); // Print the integer part of the variable
   Serial.print("."); // Print the decimal point
   // Determine the fractional part. The 10 multiplier gives us 1 decimal place.
   frac = (flowRate - int(flowRate)) * 10;
   Serial.print(frac, DEC); // Print the fractional part of the variable
   Serial.print("L/min");
   // Print the number of litres flowed in this second
   Serial.print("  Current Liquid Flowing: "); // Output separator
   Serial.print(flowMilliLitres);
   Serial.print("mL/Sec");
   // Print the cumulative total of litres flowed since starting
    Serial.print("  Output Liquid Quantity: "); // Output separator
    Serial.print(totalMilliLitres);
    Serial.println("mL");
    if (flowRate > 0) {
    digitalWrite(ledPin, HIGH);   // turn the LED on (HIGH is the voltage level)
    delay(100);
   // Replace <YOUR_API_KEY> with your EmonCMS API Key
   sprintf(push_data,
//"http://emoncms.org/input/post?json={frac:%d.%d,flowml:%d,totalml:%d}&node=Penampung
2&apikey=<YOUR_API_KEY>",       int(flowRate),       int(frac),       flowMilliLitres,
totalMilliLitres);
   Serial.printf("%s\n", push_data);
    http.begin(push_data);
    digitalWrite(ledPin, LOW);    // turn the LED off by making the voltage LOW
    delay(100);
    int httpCode = http.GET();
    // httpCode_code will be a negative number if there is an error
     Serial.print(httpCode);
     if (httpCode > 0) {
    digitalWrite(ledPin, HIGH);   // turn the LED on (HIGH is the voltage level)
     delay(100);
    // file found at server
     if (httpCode == HTTP_CODE_OK) {
     String payload = http.getString();
     Serial.print(" ");
    Serial.print(payload);
```

```
            }
 digitalWrite(ledPin, LOW);    // turn the LED off by making the voltage LOW
  delay(100);
        } else {
 Serial.printf("[HTTP] GET... failed, error: %s\n",
  http.errorToString(httpCode).c_str());
   }
   http.end();
     }
 // Reset the pulse counter so we can start incrementing again
     pulseCount = 0;

 // Enable the interrupt again now that we've finished sending output
     attachInterrupt(sensorInterrupt, pulseCounter, FALLING);
   } else if (WiFi.status() != WL_CONNECTED) {
     startWIFI();
    }}
/*
 Insterrupt Service Routine
 */
void pulseCounter() {
 // Increment the pulse counter
 pulseCount++;
}

void startWIFI(void) {
 digitalWrite(ledPin, HIGH);   // turn the LED on (HIGH is the voltage level)
 delay(100);

 WiFi.begin(ssid, password); // Connect to the network
 Serial.print("Connecting to ");
 Serial.print(ssid);
 Serial.println(" ...");
 oldTime = 0;
 int i = 0;
 digitalWrite(ledPin, LOW);    // turn the LED off by making the voltage LOW
 delay(100);

 while (WiFi.status() != WL_CONNECTED) { // Wait for the Wi-Fi to connect
     digitalWrite(ledPin, HIGH);   // turn the LED on (HIGH is the voltage level)
     delay(2000);
     Serial.print(++i);
     Serial.print('.');
     digitalWrite(ledPin, LOW);    // turn the LED off by making the voltage LOW
     delay(100);
 }
 delay(2000);
 Serial.print('\n');
 Serial.print("Connection established!");
 Serial.print("IP address:\t");
 Serial.print(WiFi.localIP()); //Send the IP address of the ESP8266 to the
 Computer
 }
```

7.WORKING

In general water flow measurement is necessary for various purposes in industrial,domestic……using various methodologies.In this project we are going to measure the water flow using water flow sensor and nodemcu .

When the water flows through the water flow sensor placed in the middle of the flow,the sensor detects the flow and is represented as emf at the output terminal of the flow sensor.The obtained EMF at the output of the sensor will be proportional to the flow rate at which the water is being consumed.The output of the sensor is connected to the nodemcu ,where the proportional emf will be converted into the flow rate in litres per hour or milli litres per second based on the coding provided to the nodemcu board. Now open the Arduino software and run the code in pc and then open the serial monitor and set the baud rate same as that

was mentioned in the code .Now the flow rate will be displayed in that monitor.In this way we can contionously monitor the flow rate by using iot .

EMF Induced in the sensor α Rate of flow

8.Practical hardware model of iot water consumption monitoring

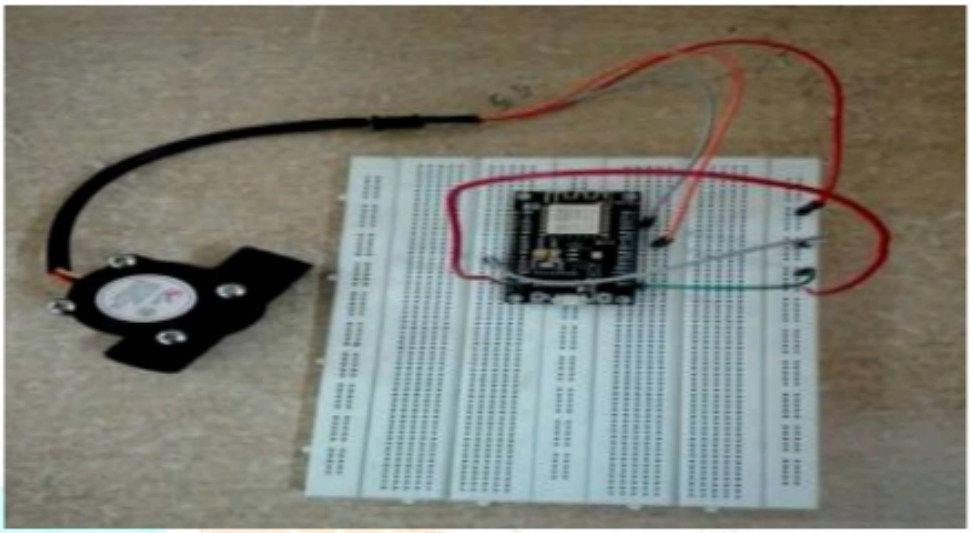


Fig:4 Hardware of water consumption monitoring

9. FUTURE SCOPE

In future, the proposed system can be used to monitor and analyze water usage of the specific water source thus require developing such logic for the application. The system can also be used to collect and study the environmental data of water source,water consumption at domestic level ,industrial level…and its surrounding area by integrating other sensor to the system. The study may include location data, water quality, temperature, humidity and various other factors.

10. CONCLUSIONS

This IoT based proposed system is used to acquire water flow details of a water source in real time from any location, any device connected to Internet. This water flow data can be used for various purposes for better management of water source.

Monitoring water flow and its consumption from remote location may be very useful when it is not possible to visit location physically every time. The system can be implemented for different sources of water by replacing sensor device suitable for the condition.


THANK YOU