Models for Plug-and-Play IoT Architectures

Alexandros Tsoupos, Mukesh Jha and Prashanth Reddy Marpu

Abstract The Internet of Things (IoT) is already counting more than 15 billion devices connected to the web and over the following years, a rapidly increasing number of businesses and individuals are expected to become a part of this industry. In this context, enabling technologies and services are needed in order to accommodate this unprecedented interest. One of the major bottlenecks during the development of IoT products and/or services has been the vast diversity and incompatibility that exists among sensors, communication, and computation/controller modules. The various modules operating on numerous communication buses and protocols requires development of platform-dependent hardware and software drivers as well as vigorous testing from the developer side. This process is nontrivial and time-consuming and redirects the focus of developers from "what" to "how" to develop. In addition, users/developers are discouraged when they are obligated to perform tedious manual configurations before they are ready to use their products. Furthermore, there is a significant heterogeneity in IoT network architectures and existing automatic service discovery and configuration protocols. The majority of these protocols have been developed for conventional computer systems and as a result, it cannot be used by resource-constrained IoT devices. For the above reasons, various models of the well-known concept in mainstream systems of Plug-and-Play (PnP), are being introduced to the embedded systems world as well, to tackle the above issues. In the following chapter, an overview of what a Plug-and-Play architecture consists as well as a survey of the state of the art is presented.

A. Tsoupos · M. Jha · P. R. Marpu (✉)
Masdar Institute of Science and Technology, Masdar, UAE
e-mail: pmarpu@masdar.ac.ae

A. Tsoupos
e-mail: atsoupos@masdar.ac.ae

M. Jha
e-mail: mjha@masdar.ac.ae

H. M. Ammari (ed.), *Mission-Oriented Sensor Networks and Systems: Art and Science*, Studies in Systems, Decision and Control 164,
https://doi.org/10.1007/978-3-319-92384-0_5

1 Peripheral Plug-and-Play (PnP) Definition and First Attempts

The Plug-and-Play (PnP) concept, in terms of peripheral device integration, can be defined as the ability of the system to automatically detect and configure internal and external peripherals as well as most adapters. In personal computers, the manufacturers quickly recognized the tedious task of having to manually configure hardware jumpers and software settings, and started to shift towards architectures that would allow simple and automatic integration of peripherals.

1.1 NuBus

One of the first attempts of PnP architecture was the MIT NuBus that was introduced in 1984, and was first standardized in 1987 [1]. In comparison to the existing architectures at that time, featuring 8-bit or 16-bit buses, the Nubus was equipped with a 32-bit backplane to accommodate future systems. The NuBus did not feature a distinct bus controller which meant that all NuBus devices participated as peers to system control functions and arbitration. Furthermore, an identification scheme was present which allowed for automatic detection and configuration of NuBus cards from the host system. The bus offered a form of geographic addressing, meaning that each available slot had a small dedicated address with which was associated. Specifically, 32-bit physical addresses were multiplexed with the data lines and this address space was shared among all the existing slots. Up to 15 slots were available, and each of them featured a 4-bit ID field with which every communication process was qualified.

On the other hand, NuBus's implemented addressing scheme along and low clock speed (10 Mhz) compared to other architectures of the time, rendered the bus slow. Especially, the bus was not suitable for newer and faster I/O devices that did not have enough local buffering capabilities.

The NuBus is perceived as one of the pioneers of PnP hardware architectures. Texas Instruments acquired the project and developed a number of LISP and UNIX systems based on the NuBus. Later, after its standardization, the architecture was used in some Apple projects (Mac II, Mac Quadras) but eventually became practically extinct when Apple adopted the Peripheral Component Interconnect (PCI) bus [2] in their products.

1.2 MSX Bus

Another PnP architecture that was introduced in the 1980s, was Microsoft's MSX [3]. MSX was developed with the aspire to become the single industry standard for home computing systems. The idea was to enable hardware peripherals, software applications, and computer systems, which is developed by different manufacturers/organization to be interoperable when they were MSX compatible.

The MSX system was based on the Zilog Z80-family of CPUs which is intended for home computing systems. MSX offered a very well-developed hardware abstraction layer which was implemented in the MSX-BIOS. This abstraction offered extensibility, peripheral independence, and instant PnP with zero user intervention. Virtual addressing was implemented using various slot/subslots which avoided any possible conflicts. The required drivers were already installed in the cards ROMs and were able to be automatically configured. The MSX was mostly popular in Japan and acted as the platform for many important Japanese game studios.

1.3 Micro Channel Bus

Last, the Micro Channel Architecture [4, 5] that was introduced by IBM in 1987, was the successor to IBM's ISA bus and proved to be the precursor to PnP systems known to current date such as the widely adopted PCI bus. The Micro Channel cards were 32-bit but also allowed 16-bit implementations for back compatibility and featured a unique 16-bit identifier which was software read. The OS/BIOS after reading the identifier successfully, proceeded with the search for appropriate device drivers. The IDs were stored in *Reference Disks* which IBM had to update in a regular basis. When a new card was inserted, and the system was unable to find corresponding drivers boot failures occurred. In the reference disks, along with the drivers further information was provided, such as the card's memory addressing and interrupts, which is crucial for the functionality of the system.

Using this information, the system could configure a new card without any intervention from the user. However, every time a new card was installed, the system changes (interrupts, etc.) had to be saved to a floppy disk which then became necessary for every subsequent hardware change. This proved to be an important design flaw, especially when the bus was used by large corporations. Therefore, although the Micro Channel was considered successful, soon after the release of the PCI bus, it became obsolete.

Current day PnP interfaces are IEEE 1394 (FireWire), Universal Serial Bus (USB), PC card (PCMCIA), and PCI including its variants such as Mini PCI, PCI Express, etc.

2 PnP Architectures

2.1 PnP Requirements in IoT

As mentioned in the chapter's introduction, the vast variety of non-standardized IoT modules has resulted in an enormous heterogeneity that hinders communication and

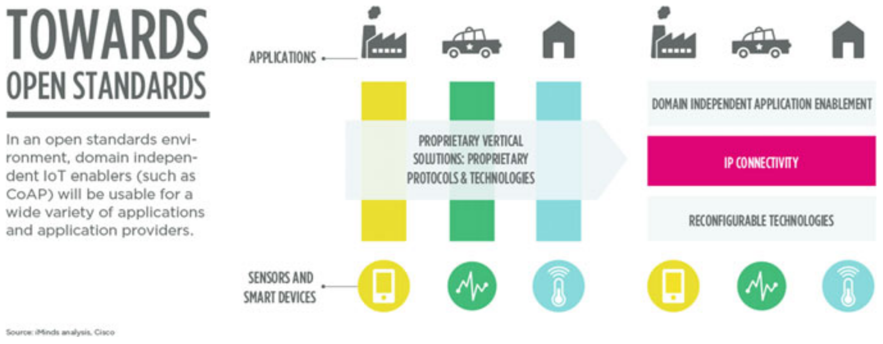

TOWARDS
OPEN STANDARDS
In an open standards environment, domain independent IoT enablers (such as CoAP) will be usable for a wide variety of applications and application providers.
APPLICATIONS
PROPRIETARY VERTICAL SOLUTIONS: PROPRIETARY PROTOCOLS & TECHNOLOGIES
DOMAIN INDEPENDENT APPLICATION ENABLEMENT
IP CONNECTIVITY
RECONFIGURABLE TECHNOLOGIES
SENSORS AND SMART DEVICES
Source: iMinds analysis, Cisco

Fig. 1 Towards open standards

interoperability among objects and architectures. Currently, the mainstream practice in IoT development is based on vertical and proprietary solutions, in the sense that a hierarchical bottom to top design (hardware–software-communication) is carried out to suit the needs of each application. This model has to be transformed (Fig. 1) to an open and horizontal structure that will act as a vessel for universal, interoperable, low-cost, and innovative solutions. IoT-enabling tools and services should be developed in a layered and easy to interconnect manner. Following this new model, a multitude of well defined and open IoT-enabling technologies will become available to anyone interested in the IoT sector.

PnP architectures are being developed with the aim of automating or reducing the complexity of the task to configure a new “thing”. Configuration, here refers to the procedure which starts when a “thing” is physically connected to a system, to the point that it is able to connect and interact in a network consisting of more “things”. The interaction within the network can be either human or machine initiated. A “thing” can be a sensor, an actuator, a communication module, or in general an embedded system with a unique identifier that can communicate in a network either standalone or through some host gateway.

Due to a number of considerations such as cost, size, and available power, things that are used in IoT projects differ significantly from modern day computing systems in terms of resources. Even though there is a multitude of controller units available in the market and one can choose the unit that best fits the application, Table 1 shows key characteristics of four controller units that are predominantly used in different types of applications. As it is expected, a higher amount of resources is available on more expensive and power hungry systems. To achieve a good balance between cost and the set of features that an IoT solution will offer, it is necessary to develop an efficient and effective IoT architecture.

This resource-limited nature of embedded IoT projects, imposes a number of constraints and challenges on the hardware and software schemes that are required for achieving PnP capabilities. Therefore, every IoT-PnP architecture needs to be “lightweight” in a series of aspects. The most important of these aspects are:

Table 1 Key parameters of popular IoT controller units [6–9]

MCU / Resource	ATmega328P	CC3200	SAM9G25	Raspberry Pi 3
Computation				
Architecture	8-bit AVR	32-bit ARM-M4	32-bit ARM A-5	64-bit ARM A-53
Max. frequency (MHz)	20	80	400	1200
Floating-point unit	–	–	–	Yes
MIPS	1/MHz	1.25/MHz	1.57/MHz	2.3/MHz
Connectivity				
Wired	–	–	USB/Ethernet	USB/Ethernet
Wireless	–	Wi-Fi/Bluetooth	–	Wi-Fi/Bluetooth
Memory				
FLASH/EEPROM	32 kB/1 kB	64kB + SD card	64MB	SD Card up to 64GB
RAM	2 kB	256 kB	32 kB on chip	1GB
Max. power consumption (W)	0.06	0.9	0.4	2
Price	2.1 $	12.1 $	7.8 $	35 $

- *Power consumption*: Numerous IoT projects are destined for battery-powered applications. To decrease battery requirements and operational time, the total power consumption of a PnP architecture has to be minimized.
- *Cost*: Cheap IoT modules have been the backbone of the industry's rapid growth. The hardware and software implementation of the architectures should not impose excessive additional cost.
- *CPU overhead*: CPUs that are selected in embedded environments usually have limited computation capabilities. The PnP service should not inflict major CPU overheads that will burden the CPUs typical operations.
- *Memory footprint*: Both program and data memories are constrained. PnP protocols have to be simple and effective.
- *Communication footprint*: In many IoT projects, excessive amount of header or information related to the PnP architecture will result in increased costs (e.g., GPRS data plans.
- *Implementation complexity*: The PnP architecture has to be easily implemented by both, hardware and software engineers.

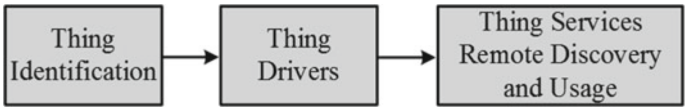


Fig. 2 The PnP process

3 PnP General Architecture

PnP can be defined as a three-step process as shown in Fig. 2. First, the newly inserted to the system thing, is uniquely identified. This preconditions that an identification mechanism is present, and it is compatible among the thing and the system. After identification, appropriate software drivers are required to interface the system with the thing. These are low-level drivers that act upon the control registers of the hardware interface to which the thing is physically attached. Last, users of the IoT network should be able to discover and use the services that are offered by the connected things in order to develop high-level applications. This process corresponds to the process called remote service discovery and usage. In the following sections, each of the above steps will be described in detail.

3.1 Thing Identification

As mentioned before, the identification of new peripherals is the first step for their integration into a system. In mainstream computing systems, such as contemporary desktops and laptops, peripheral identification is realized through custom Integrated Circuits (ICs) that are required for the operation of the interface. These ICs contains data for device identification in special purpose registers. For example, identification in the PCI bus is performed with the use of a file called *Extended System Configuration Data (ECSD)*. This file contains information about the installed PnP devices such as identification data, configuration parameters, etc. This information is read by the BIOS and through the Operating System (OS) system handlers, the identification process gets completed.

However, this approach can only be applied to resource-rich systems and not for embedded IoT devices as it is optimized for performance and neglects CPU overhead, memory footprint and/or power consumption.

Common hardware interconnects in embedded systems for interfacing things and systems are:

- Analog (voltage and current). This interconnection is used for things with analog output. In this case, an Analog-to-Digital Converter (ADC) is required to convert the analog signal to a digitized form is easily processed by the controller.
- Serial Peripheral Interface (SPI) [10]. SPI is a synchronous bus (sampled at a specific clock rate) consisting of four lines: Master In–Slave Out (*MISO*), Master

Out–Slave In (*MOSI*), Clock (*CLK*), and Slave Select (*SS*). Usually, input and output pulses are positive and negative edge sampled which means that in some cases, MISO and MOSI can be connected to the same line. The typical implementation of the SPI interface uses 8-bit messages.
- Universal Asynchronous Receiver Transmitter (UART) [10] which can be half or full duplex. As the name reveals, this is an asynchronous communication scheme between two entities. Due to the asynchronous operation, single or double buffers are required. Furthermore, the communication bit rate (baudrate) and data frame have to be configured manually to match between the transmitter and the receiver. Usually, UART is the TTL/CMOS voltage level application of the RS232/RS485 protocols which are more common in industrial and/or long communication lines. In case of half-duplex operation with no hardware flow control, UART uses only two communication lines T_x, R_x.
- Inter-Integrated Circuit (I^2C) and System Management (SM) Bus [11]. These buses feature a 7-bit addressing scheme making it possible for a total of 128 devices to operate on the same bus. A single master is allowed, which is usually a controller unit and is capable of initiating transactions with the slaves by broadcasting their address. Both, (I^2C) and SM buses require two lines (Serial Clock—SCL and SDA—Serial Data) but differ in the amount of commands that they support; the SM bus command protocol is a *subset* of the commands available in the (I^2C) bus.
- General Purpose Input/Output pins. A number of sensors interact with the controller unit through simple protocols that do not require hardware acceleration and can be implemented only in software (bit banging). Furthermore, GPIOs can be used by sensors to generate main CPU interrupts and/or to control other devices.

Although the above hardware interfaces and corresponding protocols offer simple, lightweight and in some cases fast communication, they lack device identifiers that are essential to a PnP architecture.

As the vast majority of today's sensors are manufactured using one of the above interfaces, a type of middleware which will provide an identification procedure is mandated. After the identification is complete, a set of multiplexers has to switch the peripheral to the appropriate interface bus in order to communicate with the host CPU(s).

3.2 *Thing Drivers*

After the identification of a thing, the corresponding drivers need to be installed and automatically configured. From a general perspective, there are three ways in which this can be achieved.

First, the thing drivers can be potentially stored in a nonvolatile memory on the thing itself. This is an approach that has been dismissed already by PnP technologies intended for peripheral integration in mainstream computing machines. The reasons were the additional memory required to store the drivers resulting in increased cost and the complexity deriving from the fact that the machine has to communicate with the device and download the drivers. Furthermore, this is an approach that does not allow for changes to the drivers rendering them outdated for a large span in the device's lifetime. Taking into account, the more cost-sensitive and dynamic environment of IoT projects, such a model cannot be applied during the peripheral integration procedure. The dual approach, i.e., to store the thing drivers on the controller side, is also discarded because of the enormous and continuously increasing variety of available things and hence corresponding drivers.

The current practice during driver development for IoT, is to write driver software after reviewing in detail the thing's datasheet and list of specifications. The developed drivers are written in low-level programming languages in which register manipulation and interrupt handling are required rendering them platform specific. This is a cumbersome task, since the development of a set of drivers that will be both reliable and also efficiently use the thing's functions as a repetitive process. Furthermore, this process leads to nonreusable IoT application code as even trivial hardware changes require updating the driver software.

In modern computing systems, the most common scheme that is followed in PnP peripheral driver integration is the following. Once a new device is connected to and detected by the system, an OS service will start searching local drives and/or remote repositories for the appropriate software drivers. If drivers that match the device id are found, the OS service continues with their installation. After this point, the OS is ready to fully use the peripheral.

In desktop computers, PnP capabilities were first introduced by Microsoft's operating system Windows 95. Microsoft's scheme followed the idea described earlier. The process is illustrated in Fig. 3. When a new device that supports PnP is connected to the system, the service called Plug-and-Play Manager follows a number of steps in order to install the device [12].

- After the detection of a new device, the Plug-and-Play Manager checks the hardware resources required by the device and allocates them.
- The Plug-and-Play manager checks the device's hardware ID and then searches for matching drivers the hard drive(s), floppy drive(s), CD-ROM drive(s) and finally, the Windows Update website.
- Further identification features such as driver signatures or the closest compatible hardware ID used in case multiple drivers are found.
- After security and quality checks, the Plug-and-Play Manager installs the selected driver and the OS is then ready to use the device.

While the above scheme is comprehensive and reliable, the identification and validation layers are not optimized for resource-constrained systems. Identification data are stored in formats (e.g., XML) that are optimized for performance and reliability rather than minimizing CPU overhead and memory footprint. Similarly, in the transport layer, the communication with the remote repositories is performed using rich protocols such as HTTP over TCP/IP. The implementation of the above in a resource-constrained system would be either impossible or would impose nontriv-

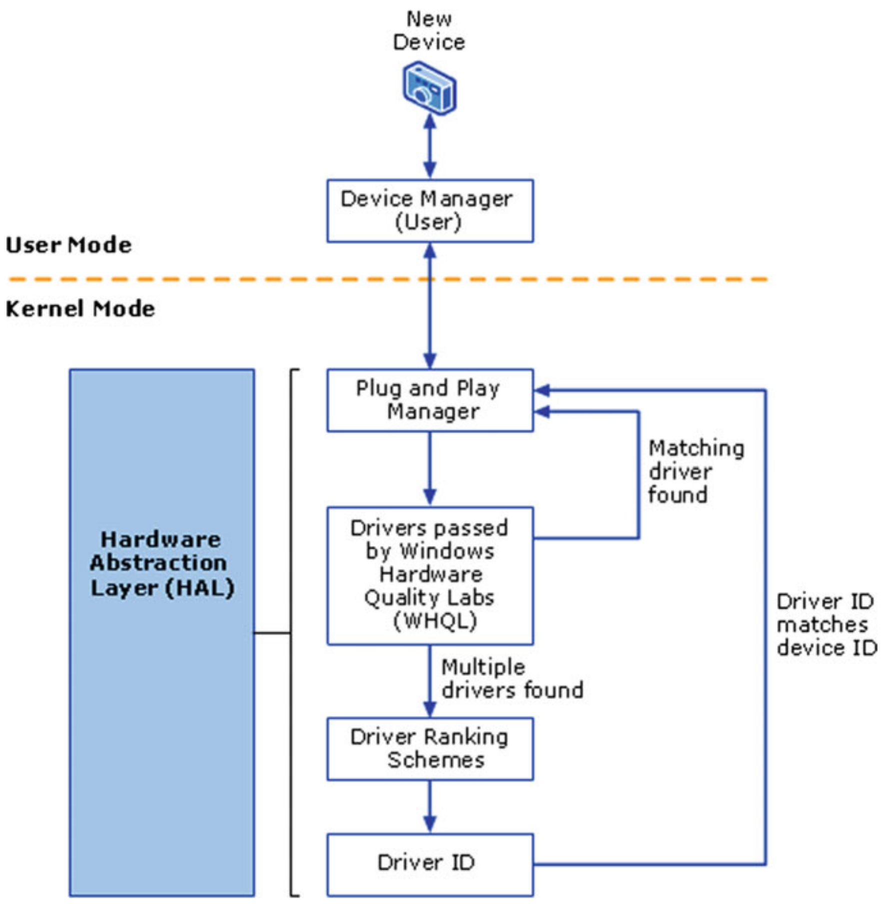


Fig. 3 Microsoft's PnP explained

ial CPU and memory overheads resulting in significant limitations to the range and quality of the developed application.

Thus, it becomes clear that in order to move towards realistic and efficient PnP architectures in the embedded environment of IoT products, simpler and more lightweight protocols that are cross-platform have to be developed while maintaining similar PnP characteristics with resource-rich systems.

3.3 Thing Network Discovery and Operation

After the thing is identified and the software drivers are installed, it is capable of data transactions with its host system. However, the end goal is to create a horizontal

PnP IoT architecture where developers will be able to write applications without a comprehensive knowledge of the underlying hardware but only of their services. For this, a scheme where things publish their available services over a network and developers are able to discover and use them is needed. In mainstream computer systems, this is realized through service discovery protocols. These protocols provide mechanisms that allow to:

- Dynamically discover devices and corresponding services.
- Network users to search and browse the available services.
- Advertise service information crucial to users.
- Utilize an available service.

3.3.1 Service Discovery and Usage in Mainstream Computer Systems

Jini [13]

Jini is a distributed system based on the idea of federating groups of users and the resources required by those users. In a simpler interpretation, the Jini architecture is constructed by a number of hardware and software components that are the infrastructure of distributed system; the Jini registrar provides unicast or multicast detection of this infrastructure (services) by the client returning a proxy object to the clients. This object besides pointers to services can also store Java-based program code that will make use of the service easier to a client. The main mechanism responsible for the communication between services and clients is called *lookup service*. Each Jini device is assumed to have a Java Virtual Machine (JVM) [14] running on it. Jini's main advantage is that it allows users to connect with services without previous knowledge of their address through the lookup service. On the other hand, the existence of the lookup service to manage the interactions between clients and services renders the architecture not suitable for large networks.

Universal Plug-and-Play (UPnP) [15]

Universal Plug-and-Play (UPnP) is a media-independent networking scheme leveraging TCP/IP and other established web protocols. It is developed by an industry consortium called UPnP Forum, which has been founded and lead by Microsoft. Taking into account, the previous discussion about software drivers, UPnP extends Microsoft Windows Plug-and-Play to devices that can communicate in a network. Every device can dynamically join a network, obtain an IP address, announce its services, and also communicates with other devices and services in the network using multicast communication. UPnP is applicable on networks that run Internet Protocol (IP) and on top, it utilizes protocols such as HTTP, SOAP, and XML [16] to accommodate the interactions between the devices. UPnPs main difference with Jini is that it can operate in a decentralized way in the sense that discovery and service advertisement are modeled as events, and are transmitted as HTTP messages over multicast User Datagram Protocol (UDP) [17]. However, this makes the network chatty consuming a significant amount of resources.

Service Location Protocol (SLP) [18]

Another protocol allowing clients to find services over a network without any prior configuration is the Service Location Protocol (SLP). SLP, as UPnP, uses multicast routing in a decentralized way, at least in smaller networks. Each Service Agent (SA)—in our case a thing—multicasts advertisement messages periodically which contain a URL that is used to describe and locate the service. In scaled-up networks, Directory Agents (DAs) exists and cache the information announced by the SAs. User Agents (UAs) can discover services by either multicast requests to the DAs or can listen directly to the announced messages in the absence of DAs.

All of the above architectures have worked well for mainstream computer networks but it is a burden for the proliferation of IoT systems as they are not resource optimized. Jini requires a full-fledged Java Virtual Machine, UPnP operates on verbose XML data representations and SLP mandates further filters due to the lack of device identifiers.

3.3.2 Service Discovery and Usage Protocols in IoT

The Open Connectivity Foundation—IoTivity

Due to the vast expansion of IoT, various company groups have started to develop service discovery and connectivity protocols designed as IoT enablers. One of the biggest, is the Open Connectivity Foundation (OCF) [19] counting more than 300 member companies where among them are Samsung Electronics, Intel, and Microsoft. Its purpose is to "accomodate the communication of billions of connected devices (phones, computers, and sensors) regardless of manufacturer, operating system, chipset, or physical layer". IoTivity [20] is an open multi-layer framework for IoT networks hosted by the Linux Foundation and acts as the reference project implementing OCFs specifications.

Constrained Application Protocol (CoAP) [21]

CoAP, as the name unveils, is an application layer protocol designed for devices with constrained resources in low-power and lossy networks. Typical applications, as defined by the protocol, are wireless nodes that run on 8-bit microcontrollers with low RAM and ROM capacities and slow, high packet error rate networks such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs) [22]. The CoAP design is optimized for Machine-to-Machine Communications (M2M). A very important aspect of CoAP is the easy mapping with HTTP which makes web integration much simpler.

CoAP has been mainly developed and standardized by the Internet Engineering Task Force (IETF) Constrained RESTful enviromnmets (CoRE) Working Group and as complete standard was proposed in 2014. The protocol is maintained and updated by IETF working groups to the current day. CoAP provides resource/service discovery and usage as the aforementioned protocols and is based on the REST model: servers/devices make services available under a URL and clients access these resources through commands such as GET, PUT, POST, and DELETE. These command messages are kept as small as possible to support the use of the UDP protocol in the transport layer and to avoid message fragmentation.

4 Plug-and-Play Models for IoT Applications

4.1 *MicroPnP*

MicroPnP or μPnP [23–26] developed by the research group of iMinds-Distrinet in University of Leuven, Belgium is one the first complete PnP platforms specifically designed for the IoT industry. μPnP presents a holistic Plug-and-Play architecture including custom hardware for peripheral identification and integration, platform agnostic language for device driver development, and a network architecture for automatic thing discovery and usage.

4.1.1 μPnP Thing Identification

The architecture uses a simple hardware solution to map a large space of addresses to various peripherals. The custom-designed PCB of μPnP contains a set of monostable multivibrators that are capable of generating timed pulses whose length depends on a resistor and a capacitor. On the other side, every μPnP compliant peripheral embeds a unique set of resistors which in conjunction with the fixed value capacitors on the μPnP PCB creates a unique train of pulses. Specifically, four short pulses are generated and each of them is mapped to a single byte value and finally, results in a 32-bit address space. This allows for more than 4 million unique device identifiers. All μPnP peripheral identifiers are mapped to an open online global address space. Identifiers become permanent only after software drivers are integrated in the repository. After the inserted peripheral is identified, it is directed to the appropriate communication bus through a multiplexer and is ready to be utilized by downloading the software drivers.

In [27], the authors presented another model for thing identification. In this case, instead of digitizing the length of pulses generated by multivibrators, a voltage divider technique is applied (Fig. 4). Specifically, the identification resistors (R_{id}) of the thing

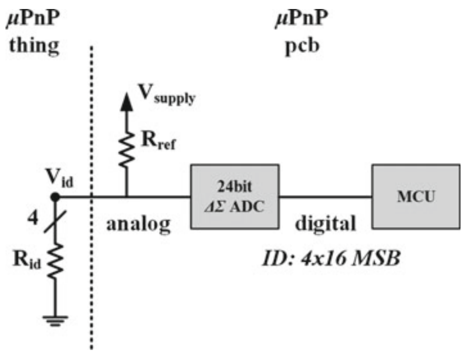


Fig. 4 Identification technique of μPnP v2.0

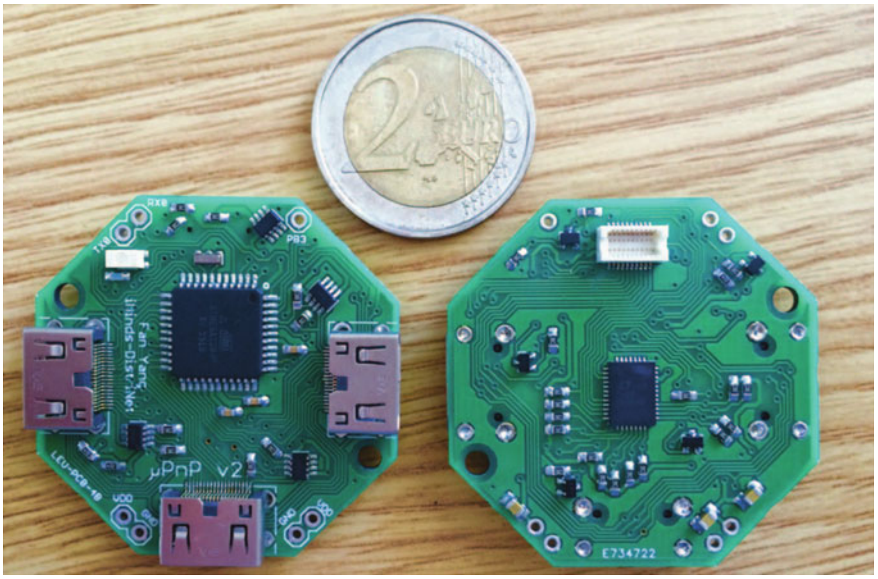


Fig. 5 μPnP peripheral identification hardware v2.0

are connected through a reference resistor (R_{ref}) to the supply rail. An ADC converts the divided voltage (R_{ref}) to a number of bits. With the use of a $\Delta\Sigma$-type ADC, high resolution can be achieved. The authors specify that for 24-bit ADC, the 16 most important bits can be used as a unique tag for things. This creates a 4×16 64-bit address space for thing identification.

Furthermore, both identification schemes require significantly lower energy than a typical USB during the identification process. The implementation of the μPnP v2.0 controller board is shown in Fig. 5.

4.1.2 μPnP Thing Drivers

To achieve multi-platform driver development, μPnP has developed a high-level domain-specific language (DSL). The run-time environment links the DSL with native hardware libraries to the underlying physical interconnects such as ADC, SPI, I2C, etc. The language is event based in order to accommodate the interrupt-driven nature of IoT software.

Platform independence is achieved by compiling the DSL into bytecode instructions that can be interpreted by the μPnP execution environment. This technique is inspired by and similar to Java virtual machines. The various abstraction layers of μPnP's run-time environment are shown in Fig. 6. Five separate elements can be distinguished. The *peripheral controller* interfaces with the μPnP identification PCB and identifies the inserted peripheral. The *driver manager* communicates with the

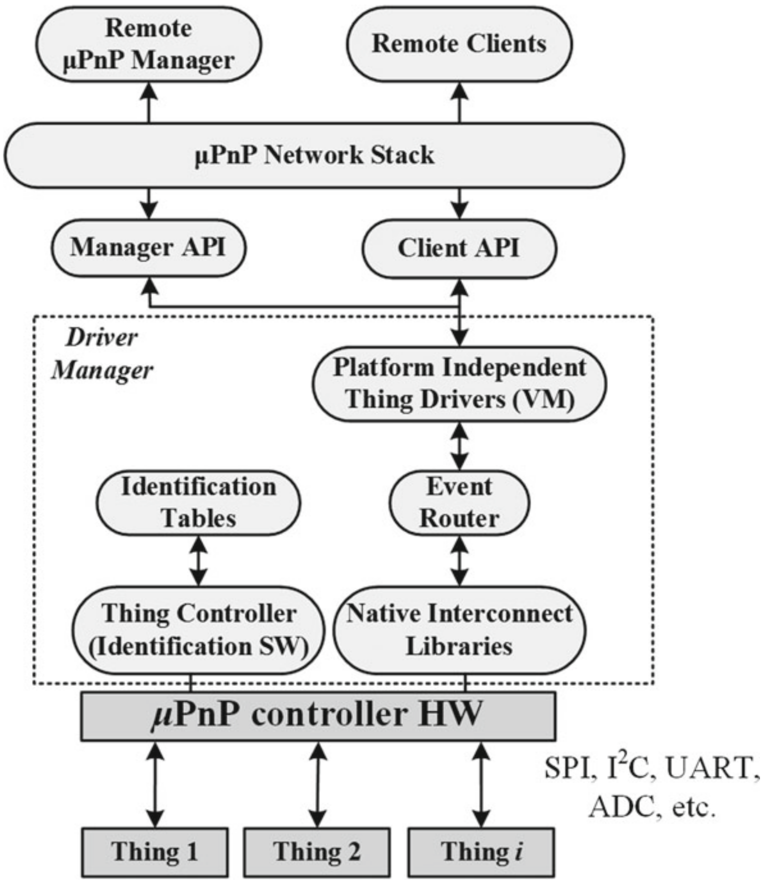


Fig. 6 μPnP execution environment

peripheral controller and gets informed about the connected things and the drivers that are installed. Furthermore, it is responsible for searching and installing available drivers for the newly identified things. A stack-based *virtual machine* executes the bytecode instructions derived from the interpretation of the DSL. Hardware-specific *native libraries* implements the low-level communication with the things. Finally, the *event router* routes the events coming from the DSL, the native libraries, and the software stack. The events are handled asynchronously and do not collide. This is achieved with the application of a FIFO queue for regular events, and a priority queue for error messages.

The thing drivers are installed in the μPnP thing through an entity called Driver Manager. Once the thing is identified, the driver installation process starts with a *driver installation request* towards the anycast address of μPnP server. If a software driver that matches the thing ID is found, the drivers gets downloaded to the μPnP system. Furthermore, as described in the previous paragraph, the Driver Manager is allowed to query the connected μPnP things about the installed drivers. This is realized by a *driver discovery* message from the host to the thing which awaits a *driver advertisement* message from the thing to the host. The manager is also allowed to remove software driver from a thing using a *driver removal* message. The removal is complete when the thing responds with *driver removal acknowledgment* message.

4.1.3 μPnP Service Discovery and Network Architecture

μPnPs networking scheme features automatic and remote thing discovery and usage like protocols is described earlier and are used in mainstream computer systems. The architecture consists of mainly three software entities. The first entity is the μPnP Thing software which runs on the resource-constrained local machine and allows for automatic peripheral identification. Second, is the μPnP client software which can run on an embedded device or a standard platform and realizes the remote discovery and usage of peripherals that may exist on any node of the network. Last, is the μPnP Manager which runs on server-class machine and is responsible for the remote dispatch and deployment of the thing drivers.

To leverage existing network technologies such as Ethernet and Wi-Fi, the entities are interacting on the network layer through IPv6 over UDP. The thing discovery and usage is realized with three types of communication.

- Unsolicited peripheral advertisements. This source of this announcement is the thing's unicast IPv6 address and contains a set of fields describing it. The unsolicited peripheral advertisements are multicasted to the set of μPnP clients every time a new thing becomes available.
- Peripheral discovery messages. These messages are issued by the clients containing the type of peripheral that is searched. The destination of the messages is the multicast address of all the μPnP things with the specific type of peripheral.
- Solicited peripheral advertisements. These messages are sent in response to peripheral discovery messages. They contain the same information as the unsolicited advertisements but are destined to the unicast address of inquiring μPnP client.

μPnP clients allowed two types of interactions with the things for data production: (a) single value reading and (b) data streaming from the service provider to the client. Writing data to a thing is also supported in case the thing has actuator capabilities.

The above transactions are realized with the following messages:

- *Read* allows a μPnP client to read single value from a peripheral. The μPnP thing that the peripheral is connected to responds with a *data* message which contains the result.
- *Stream* messages are send from clients to things when a client wants to subscribe to a continuous stream of data. In this case, the thing responds with an *established* message which contains the multicast address that the client should join. *data* messages are send continuously from the stream address and a *closed* message is broadcasted if the stream stops to inform all the μPnP clients.

- The control of peripheral is achieved with a *write* message. This message is sent from a client to a thing. If the write process is completed successfully, an *acknowledgment* message is given as response to the client.

4.2 IEEE 1451 Standard [28]

The IEEE1451 is a family of "smart transducer" interface standards developed by the Institute of Electrical and Electronics Engineers (IEEE) and was first released on 1997. A smart transducer is defined as the integration of an analog/digital sensor or actuator, a processing unit, and a communication interface. According to this definition in Fig. 7a, we can see the structure of a smart transducer. It consists of (1) sensors/actuators, (2) signal conditioning and/or data conversion circuits, (3) host processor, and (4) network communication. The communication paths in this structure are two-way as data can flow from the transducer to the network in the case of a sensor or from the network to the transducer in the case of an actuator.

An IEEE1451 smart transducer should have features like self-identification, self-description, self-diagnosis, self-calibration, location-awareness, time-awareness, data processing, standard-based data formats, and communication protocols. The IEEE1451 aims to achieve the above with the design and integration of *Transducer electronic data sheets or TEDS*. The architecture is shown in Fig. 7b. The modules that constitute the architecture are: (1) a Network Capable Application Processor

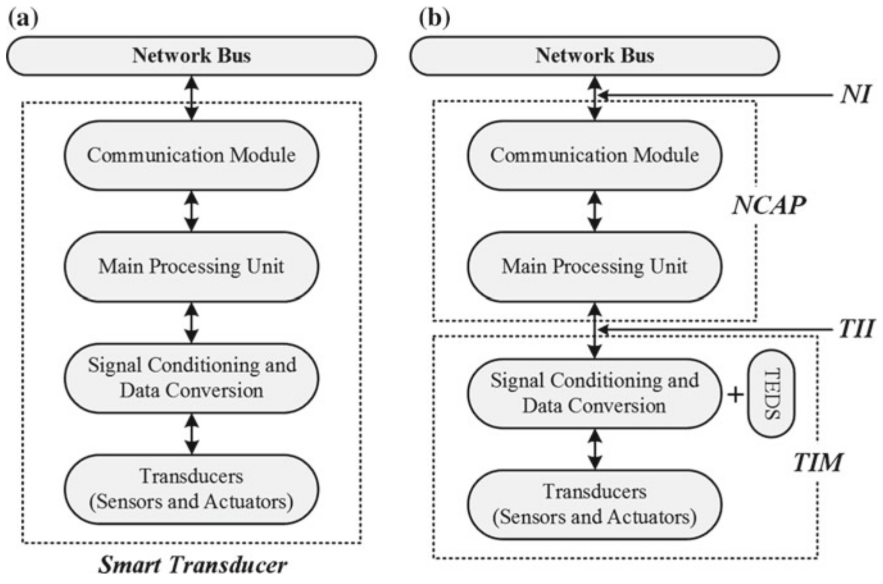

(a)
Network Bus
Communication Module
Main Processing Unit
Signal Conditioning and Data Conversion
Transducers (Sensors and Actuators)
Smart Transducer
(b)
Network Bus
NI
Communication Module
NCAP
Main Processing Unit
TII
Signal Conditioning and Data Conversion
+
TEDS
TIM
Transducers (Sensors and Actuators)

Fig. 7 **a** A smart transducer model; **b** This architecture adds TEDS and the system partition into

(NCAP), (2) Transducer Interface Module (TIM), and a (3) Transducer-Independent Interface (TII) for communication between (1) and (2). The TII is defined by a communication medium and a data transfer protocol with messages like read, write, read, and write, etc. The smart transducer can be connected to the network through any common Network Interface (NI).

The key feature of IEEE1451 is the TEDS. The TEDS can be stored in a non-volatile memory space attached to the Smart Transducer Interface Module (STIM) containing necessary information for the host system to interface with the transducer, such as identification, calibration, and correction data. Also, a virtual TEDS can be implemented, allowing legacy sensors and transducers without any storage space to be included in the standard. Four kinds of TEDS are *mandatory* for the application of the standard, and are as follows:

- Meta TEDS.
- TransducerChannel TEDS.
- PHY TEDS.
- UserTransducerName TEDS.

Optional TEDS includes Calibration TEDS, TransferFunction TEDS, Location and title TEDS, and Frequency Response TEDS.

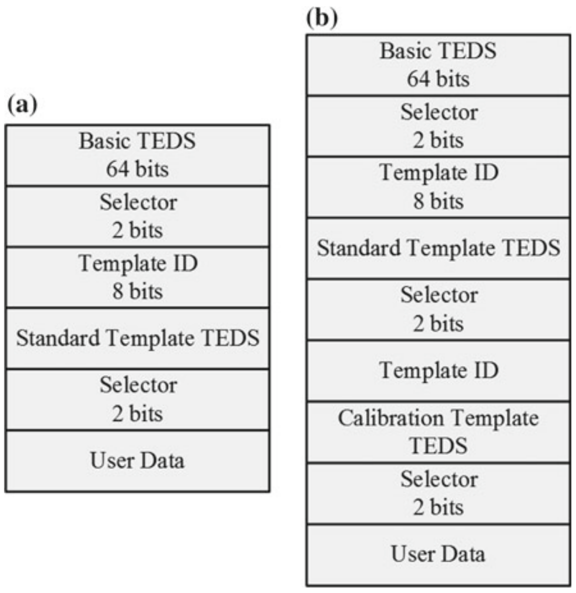
Since 1997, the standard has developed and improved in order to accommodate modern miniature sensors and actuators as well as different communication/network protocols. The full list of released protocols is:

- 1451.0-2007 Common Functions, Communication Protocols, and Transducer Electronic Data Sheet (TEDS) Formats
- 1451.1-1999—Network Capable Application Processor Information Model [29]
- 1451.2-1997—Transducer to Microprocessor Communication Protocols and TEDS Formats [30]
- 1451.3-2003—Digital Communication and TEDS Formats for Distributed Multidrop Systems [31]
- 1451.4-2004—Mixed-Mode Communication Protocols and TEDS Formats [32]
- 1451.5-2007—Wireless Communication Protocols and Transducer Electronic Data Sheet (TEDS) Formats [33]
- 1451.7-2010—Transducers to Radio-Frequency Identification (RFID) Systems Communication Protocols and Transducer Electronic Data Sheet Formats [34].

4.3 The TEDS Structure

The TEDS are encoded using specific templates to maintain a balance between the provided transducer information and the amount of memory that needs to be occupied. In the IEEE1451.4 standard, the TEDS is defined as multisection template. These sections are chained together to form a complete TEDS (Fig. 8). The first

Fig. 8 TED examples



section is the basic TEDS and contains essential identification information. Depending on the application and the complexity of the transducer, further sections can be followed. The type of the following section is indicated by 2-bit selectors. The last section, is an open user area where further information of instructions that are not defined in the template can be given.

Basic TEDS

The basic TEDS has a length of 64 bits. The first 14 bits are reserved for the Manufacturer ID, 15 bits for the Model Number, 5-bit character code for the Version Letter, 6 bits for the version number, and 24 bits for the serial number of the device. The assignment of Manufacturer IDs and other binary data are provided in ASCII files either from IEEE or from the manufacturers.

Several researchers have proposed modular PnP-like models based on the IEEE1451 standard for various applications [35–37]. In [38], the authors have leveraged the the standard in order to create a thermal comfort sensing system for buildings. In [39], a configurable Wireless Sensor Network (WSN) is developed based on the IEEE1451 and a Complex Programmable Logic Device (CPLD). The authors have used the CPLDs high parallel throughput and the interoperability offered by the IEEE1451 standards to build a flexible and reconfigurable water quality monitoring WSN.

In [40], a detailed example of how the IEEE1451 standard was practiced for the needs of an application is presented. The transducer was an electrogoniometer, which is a transducer crucial to physiotherapy applications, is designed based on the IEEE1451 standard. Specifically, IEEE1451.2 and IEEE1451.0 were implemented.

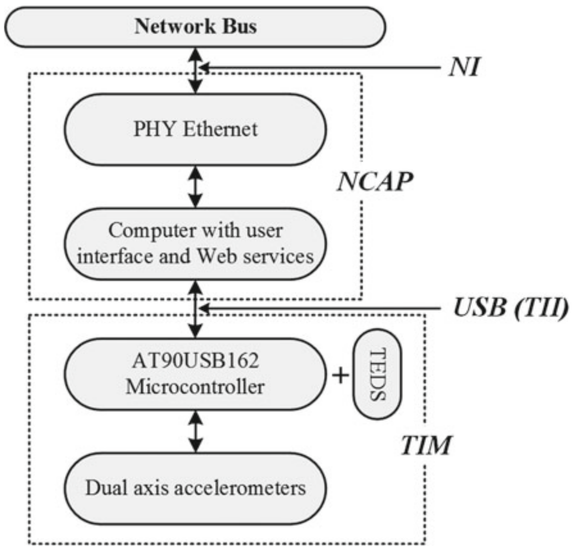

Network Bus
NI
PHY Ethernet
NCAP
Computer with user interface and Web services
USB (TII)
AT90USB162 Microcontroller
+
TEDS
TIM
Dual axis accelerometers

Fig. 9 Block diagram of IEEE1451 compliant electrogoniometer

The block diagram illustrating the implementation of the sensor is shown in Fig. 9. The STIM consists of two dual-axis accelerometers as sensing elements and an Atmel AT90USB162 microcontroller. The embedded Microcontroller Unit (MCU) is programmed as a STIM using C and the TEDS is stored into its nonvolatile memory (FLASH). Furthermore, it features integrated full-speed USB peripheral for communication between the STIM and the NCAP. IEEE1451.2 is implemented on top of the USB interface to render the sensor IEEE1451 standard compliant.

Specifically, the interfacing sequence can be described as the following. Once the STIM is connected to the NCAP, the STIM sends a *Tim Initiated Message* to annouce its existence. Next, the PHY-TEDS information is announced to the NCAP using the Publish–Subsribe method. Several values such as, TIM identification, Communication Module ID-Type-Name-Object, STIM Channel numbers-ID-name, etc. Furthermore, the NCAP handed the TEDS information including Meta TEDS, TransducerChannel TEDS, User's Transducer Name TEDS, Manufacturer ID, Version of TEDS, Number of Channel, and Serial Number. Finally, the NCAP send the commnad to acquire the sensor data.

The NCAP in this case is developed with the Java Development Kit and the Eclipse IDE, and it operates on a standard commercial laptop computer.

Reconfigurable Wireless Sensor Networks (WSNs)

In [41], the authors present a reconfigurable WSN with PnP capabilities regarding the network architecture of the testbed. The proposed architecture allows automatic configuration (Plug) of the network by utilizing a Zeroconf protocol that sets up a multi-hop network. Furthermore, reconfiguration and experimentation (Play) is achieved on the basis of RESTful interactions with each node. The node is composed of configurable transceiver(s), configurable/modular protocol stack, and a monitoring/control module.

The configurable transceiver has to be low cost/power and support parameter reconfiguration. In the author's implementation, two transceivers were used for dual-band communication. The first one was the reconfigurable TI CC1101 operating at the 868 MHz RF band and the AT86RF231 operating at 2.4GHz. The latter, is 802.15.5 compliant which means it is suitable for low-power 6LoWPAN communication.

A configurable/modular protocol stack is selected for efficient development and experimentation on existing protocols. Furthermore, the configurable/modular stack has to reconfigure the management network making each device a uniquely addressable and accessible network thing. The configurable/modular platfrom used was the CRime stack [42] and Contiki [43] 6LowPan/IPV6 stack was used for the management network. This management network stack is based on the Routing Protocol for Low-power and Lossy networks (RPL) [44]. RPL is a protocol for automatic network discovery and configuration. A dual-stack Contiki implementation is leveraged to run both protocols in parallel.

The monitoring and control module operates using CoAP which is an HTTP-like protocol redesigned for devices with constrained resources, as described in previous sections. A set of CoAP handlers enables system users to remotely configure and operate the testbed (*Play*). Specifically, the CoAP over UDP handlers (messages) that were developed allows the user to perform the following interactions:

- Remote monitoring and diagnosis of the system.
- Remote parameter tuning.
- Over the air software updates and upgrades.

Reconfigurable Wireless Sensor Nodes

Mikhaylov and Huttunen [45], Mikhaylov and Paatelma [46] introduces a type of PnP nodes for a Wireless Sensor and Actuator Network (WSAN). The concept that the authors want to achieve in this work is the fully modular development of a WSAN using modules which can virtually be anything; ranging from power sources, wireless/wired communication hardware, and controller units to sensors, actuators, encryption devices, localization hardware and/or additional memory. After physical connection of the available modules, the Main Processing Unit (MPU) should be able to identify all attached modules and download the necessary drivers using local or network connection.

The proposed hardware architecture for achieving automatic peripheral identification and integration is presented in Fig. 10. A new interface called Intelligent Modular Periphery (IMP) Interface (IMPI) is defined and is responsible for interfacing the peripherals to the MPU. The IMPI can be disseminated in functional terms in (a) the power supply lines which consist of the input voltage, output voltage and ground (Vin, Vout, GND), (b) the IMP bus lines that are reserved for device identification and control, and (c) underlying interface buses that the peripherals may require.

A Module Control and Identification Unit (MCIU) stores all the required information about the module as well as about the peripherals hosted by the module. The

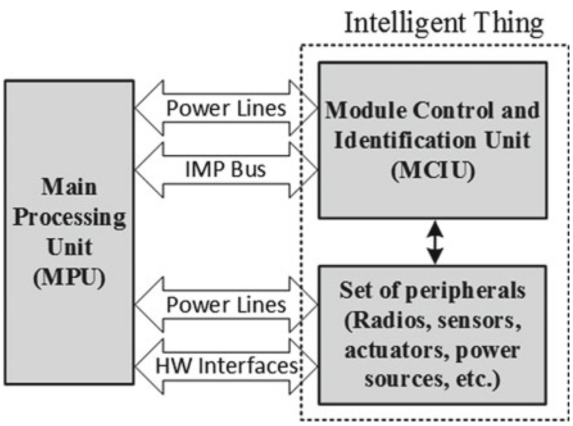

Intelligent Thing
Main Processing Unit (MPU)
Power Lines
IMP Bus
Power Lines
HW Interfaces
Module Control and Identification Unit (MCIU)
Set of peripherals (Radios, sensors, actuators, power sources, etc.)

Fig. 10 Hardware architecture

MCIU is accessed through the IMP bus and can also provide rudimentary control over the peripherals such as power management. Physically, the MCIU can be a microcontroller, a PLD, or any other logic device with similar functionality.

The IMP bus is implemented by daisy chaining the well known and ubiquitous SPI bus. Through the IMP bus, the MPU first discovers the total number of connected peripherals. Then, it downloads the peripheral description data (PDD), which consists of the Peripheral Connection Descriptor (PCD) and the Peripheral Service Data (PSD). The PCD contains the required data for the MPU to map the particular communication interfaces to specific modules and peripherals. The PSD on the other hand, contains information such as name, identifier, SW drivers, calibration coefficients, etc. Using these information, the MPU can automatically discover and use the attached peripherals.

On the software side, the dynamic underlying hardware requires a complex architecture in order to become fully functional and efficient. The proposed architecture is presented in Fig. 11. The main component of this middleware is called Resource Manager and has three major building blocks. The first one, is the Module manager, being responsible for low-level operations such as identification of peripherals and modules and interrupt prioritization. The second building block is the Communications manager and its task is to handle the communication of the node with the network. This manager asserts the existing communication transceivers and decides which must be used and under which parameters. Among the responsibilities of the Communications managers are also discovery of devices and networks, mapping/translating of network addresses, etc. Last, the Applications manager supervises and controls the launch/stop of every available application or service. The Applications manager also acts as a broadcaster of the node services in a network scheme.

The PnP Web Tag

As it was mentioned in previous sections, the current required knowledge for someone to develop a full-stack IoT application is low-level embedded programming, networking mainly using low-power protocols and transceivers and web integration.

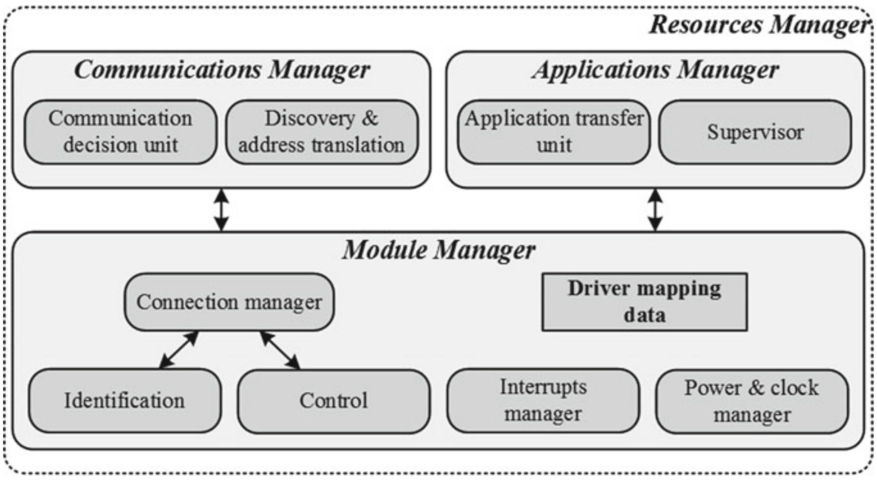


Fig. 11 Software architecture

These multidisciplinary skills hinder a lot of individuals and companies with low resources from entering the industry. In [47], the authors aim to address this problem by developing a PnP programming model for connecting IoT devices to the web.

The tool that is developed has many cores:

- A parser, that identifies IoT services in HTML pages. The parser is implemented as a pure client-side JavaScript. The parser enables to support the new PnP web tag.
- An HTML instrumenter, that refreshes in a dynamic way the HTML page as new data arrives. The instrumenter is configured by the parser and is also realized in pure-client JavaScript. Furthermore, it sends commands to the IoT devices from the JavaScript applications.
- A proxy server that acts as the interconnection between the instrumenter and the CoAP IoT services. It is implemnted in the IoT network gateway, and it bridges the CoAP protocol and the WebSockets protocol that is used by the HTML and JavaScript elements.
- JavaScript allows the web developers to ignore the low-level embedded programming and build complex sensing and control software.

PnP transducers in Cyber-Physical Systems

However, in modern IoT networks which may contain thousands of things, the added cost and the increased hardware complexity that is imposed by the standard, poses a major burden to its proliferation and widespread establishment.

Specifically, the standard defines 16 TEDS templates. The number of transducers available is increasing exponentially over the last years. A new template, requiring by the user full knowledge of the complex standard has to submitted for a new transducer to be included in the protocol. Furthermore, although IEEE1451 is an

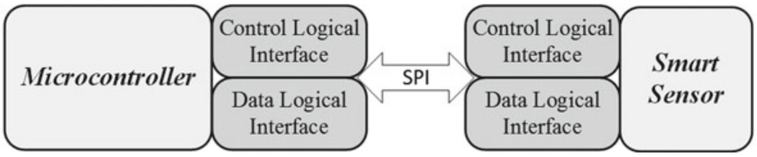


Fig. 12 Logical interfaces in the Plug-and-Play architecture

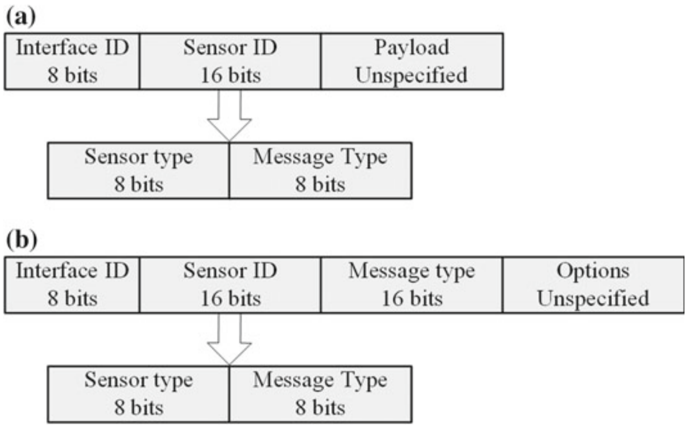


Fig. 13 **a** Data frame structure, **b** Control frame structure

open standard, a large number of these transducers contain proprietary information leading to proprietary TEDS. Second, the standard includes byte-oriented messages, thus requiring significant amount of memory for resource-constrained devices and causing nontrivial CPU overheard.

Taking into account the above, the authors in [48], proposed a new transducer/interface independent and lightweight method for PnP transducers based on the SPI bus. Over the physical SPI interface, two virtual interfaces are defined: a control and a data one Fig. 12. The physical lines are distinguished with the use of headers in the byte frames during communication. As shown in Fig. 13, the control frame communication structure includes a 2-byte message type to distinguish it from the data frame structure. With this 2-byte representation 65536 different messages can be generated, a number that can support a vast majority of applications.Please provide better clarity in the sentence “With this ... applications” and amend if ncecessary.

The identification of the modules from the main processor is implemented in three steps. First, the controller sends a Description Request message to each available slave bus. After sending the message, the controller expects a Description response message. If this message is received, the master controller records the specific slave SPI port as occupied and transmits a Description ACK message.

Plug-and-Play Software Components in Industrial Cyber-Physical Systems

In [49], the authors are using a service-oriented software architecture in order to achieve PnP in industrial systems. Specifically, a paradigm transformation is targeted,

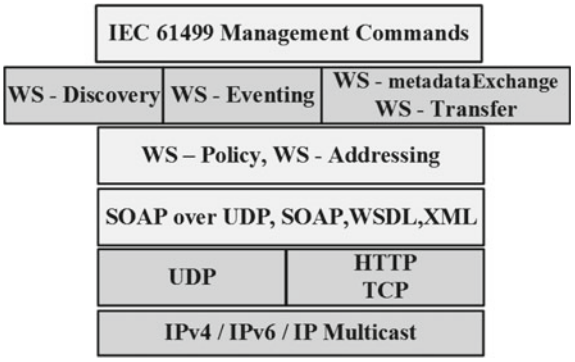

IEC 61499 Management Commands
WS - Discovery
WS - Eventing
WS - metadataExchange
WS - Transfer
WS – Policy, WS - Addressing
SOAP over UDP, SOAP,WSDL,XML
UDP
HTTP
TCP
IPv4 / IPv6 / IP Multicast

Fig. 14 Device profile for web services for Plug-and-Play software services in iCPS

from the current practice where software engineers have to develop the required software for each device and then integrate it to the system, to a model that software engineers can use directly services that are published from the devices to create high-level applications ready for system integration.

To implement such an architecture, the first step is to define the protocols and operating blocks for system level modeling. The authors propose IEC 61499 function blocks [50] for the system level modeling and Web Service Description Language (WSDL) [51] for the interface protocols.

WDSL is a language for accessing web services and is based on the Extensible Markup Language (XML). A WDSL document can define a number of elements required for service providers and users to communicate. Such elements are data types, messages, portType, binding, and services.

The above are used to render the available functions of the installed devices as callable services. For automatic service discovery, the authors select a WS-discovery type protocol which is called Device Profile for Web Services (DPWS) [52]. In WS-discovery type discovery protocols, the system services are stored dynamically in distributed registries. The complete DPWS protocol stack is shown in Fig. 14. It utilizes WS protocols based on SOAP, WSDL, and XML architectures. UDP or HTTP/TCP over IP is used in the transport layer. To enable PnP, IEC61499 management commands are used in the application layer. The typical messages used for service discovery are Hello, Probe, Bye, Resolve, Put, Get, Create, and Delete.
Generic Sensing Platform

Finding the right off-the-shelf platforms/devices for an IoT system can be a challenging and time consuming task. However, the design cost would be minimized if reconfigurable and efficient (in terms of power, size, and communication protocol compatibility) generic sensor node/platforms were available for IoT design.

In [53], the authors recognize the need of modularity and interoperability between IoT devices and sensors, and proposed a reconfigurable RFID (Radio-Frequency Identification) sensing tag as a Generic Sensing Platform (GSP) featuring PnP capabilities. RFID is a technology that suits IoT applications well as it offers low-power consumption and small size. On the other hand, RFID tags are significantly constrained in terms of sensing, computing, and data logging capabilities. Moreover, RFID sensing tags have to be accompanied by an RFID reader to be able to operate and sense.

The authors present an approach for the design of a Gen-2 [54] compatible and semipassive RFID GSP-tag. The GSP-tag is designed to accommodate a variety of sensors, multiple sensing channels, and PnP. Furthermore, the tag can operate in two modes (1) continuous data transmission mode (online) and (2) data logging mode (offline). The first three memory blocks of the user memory have been reserved and act as configuration bytes.

Specifically, the GSP-tag consists of two fundamental building blocks. An analog front-end and a digital core as shown in Fig. 15. The analog front-end contains the 915 Mhz meandering antenna as well as the L-C matching network and the modulation–demodulation circuitry. The analog front-end is power passive, so the operation of the modulation–demodulation depends on the reader's transmitted power. The digital part is implemented using an ARM Cortex-M3 microcontroller and it is battery powered. The MCU is responsible for acquiring the sensed data from the peripheral devices and performing the digital baseband communication through Gen-2 protocols with the RFID reader.

The Serial Shipping Container Code (SSCC-96) [55] EPC standard was used to perform data transmission from different channels and also act as GSPs identification mechanism. The authors as there is no global standard for RFID sensing applications, modified the EPC tag standard to provide identification for the GSP. Fields such Header, Filter, Partition, Company prefix, and Serial No. comprises a typical EPC ID. Among these, the Serial No. represents a secondary identity of the tag. Therefore, this field was modified in order to carry the variable sensor data. With this modification, automatic identification and sensor data transmission as well were achieved.

A Scalable and Self-Configuring Architecture for Service Discovery in the Internet of Things

In [56], the authors recognize the need for an architecture capable of accommodating billions of IoT nodes with minimum human intervention and proposed a Scalable and Self-Configuring Architecture for service discovery. In the paper, it is stressed that a service discovery protocol should enable communication between (1) things that are concentrated and for example, belong at the same subnetwork and (2) things that can operate within a broader scale and multiple subnetworks. Furthermore, such a protocol needs to be scalable taking into account the rapidly increasing number of devices.

The enabler which the authors propose to render the above possible is a Peer-to-Peer (P2P) network with zero-configuration (Zeroconf) mechanisms at the local scale. A dedicated boundary node, called a "IoT Gateway", acts to gather information about the resources of the locally attached nodes and create a Resource Directory (RD). This information is stored and can be accessed by other clients among the P2P network allowing for automatic Service Discovery (SD). Such a server-free approach is scalable and makes the performance of the service discovery to depend only on the size of the IoT network. To avoid application specific constraints, the architecture is designed to consist of format-of-service and resource-descriptor agnostic components.

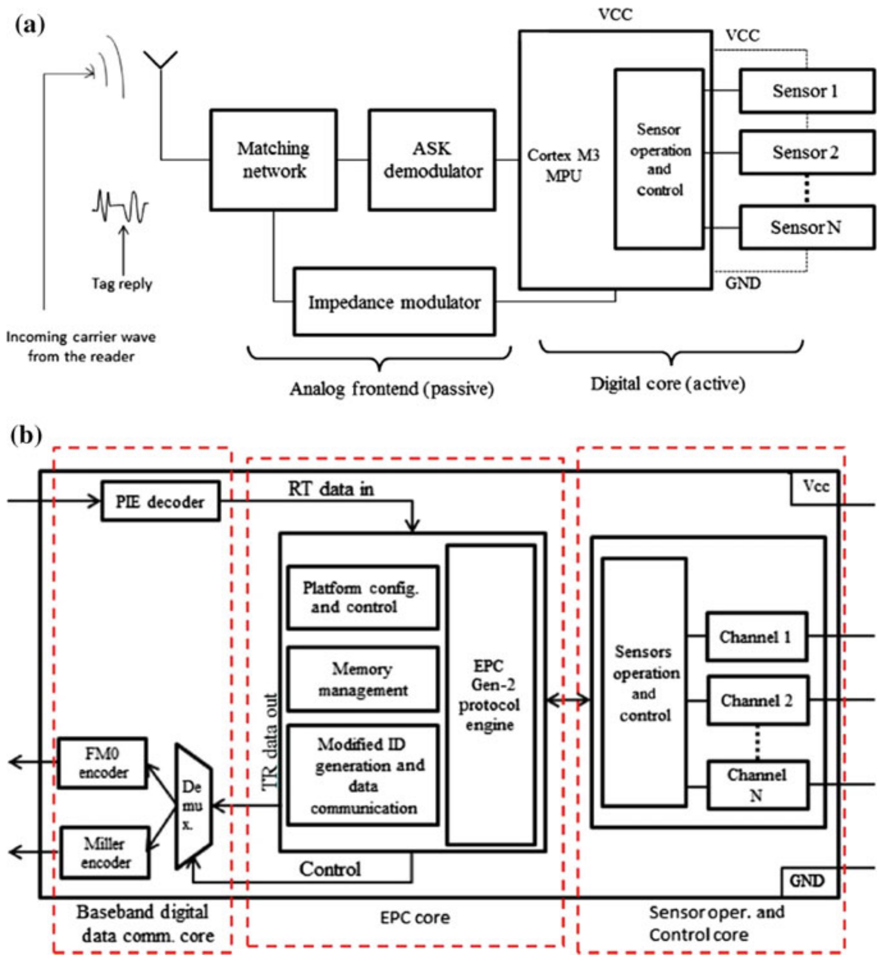

(a)
Incoming carrier wave from the reader
Tag reply
Matching network
ASK demodulator
Impedance modulator
VCC
VCC
Cortex M3 MPU
Sensor operation and control
Sensor 1
Sensor 2
Sensor N
GND
Analog frontend (passive)
Digital core (active)
(b)
PIE decoder
RT data in
Platform config. and control
Memory management
Modified ID generation and data communication
EPC Gen-2 protocol engine
TR data out
FM0 encoder
Miller encoder
De mu x.
Control
Vcc
Sensors operation and control
Channel 1
Channel 2
Channel N
GND
Baseband digital data comm. core
EPC core
Sensor oper. and Control core

Fig. 15 Implemented GSP-tag on a PCB with off-the-shelf components

The IoT gateway is responsible for providing to the nodes service discovery, caching, proxying, and access control functions. The *Proxy Function* of the gateway is performed on the application layer using CoAP. According to CoAP specifications, the gateway may act as a CoAP origin server and/or proxy. A CoAP endpoint is defined as an origin server when the resource has been created locally at the endpoint. A proxy endpoint, implements both the server and client side of the CoAP, and forwards requests to an origin server and relays back the response to the inquiring node. A proxy may also be capable to perform caching and protocol translation.

An IoT gateway architecture can be distinguished in three elements as shown in Fig. 16. Specifically, the architecture consists of the following:

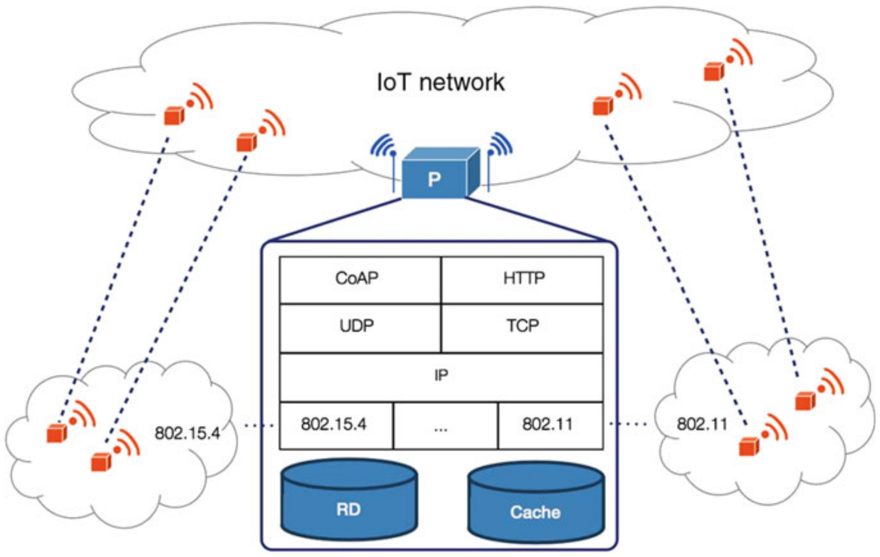


Fig. 16 Implemented GSP-tag on a PCB with off-the-shelf components

- An IP gateway capable of managing IPv4, IPv6 connectivity among things in various networks.
- A CoAP origin server that can be leveraged by the CoAP clients to post resources which are going to be maintained by the server.
- An HTTP-to-CoAP translation service for accessing services and resources are available in an internal constrained network.

In the local environment, Zeroconf is used to automatically detect and configure the new nodes that join the network. The implemented SD protocol supports in general two cases for its application:

- A new thing that joins the network publishes new services.
- A client thing, already existing in the network, discovers available services offered by other connected things.

On the first case, the process followed is different depending on whether the thing is a CoAP origin server or a server client. When the thing is CoAP server it can be queried directly about its services. On the other hand, when it is a CoAP client, it pushes the information about the available services to the IoT gateway, which acts as a RD.
Integrating Transducer Capability to GS1 EPCglobal Identify Layer for IoT Applications

The issues of resolving vast heterogeneity among IoT things is also addressed in [57]. The authors propose the application of the Electronic Product Code (EPC) global Identify Layer and IEEE1451 for building a uniform IoT architecture. The uniformity relies on the representation of the raw data collected from the sensing elements in standardized format and the PnP capabilities offered by the IEEE1451 standard.

EPCglobal is a set of standards for sharing data within and across organizations while IEEE1451, as discussed previously, is a set of standards for communication betwwen smart transducers and networks. The authors propose the adoption of IEEE1451 Smart Transducer Standard and its integration with an extension of the GS1 EPCglobal architecture.

GS1 EPCglobal architecture is utilized by numerous supply chain management systems to create track and trace applications through RFID tags. The GS1 EPCglobal is a three-layer architecture and consists of the: (1) identify, (2) capture, and (3) share layer. The identify layer gathers identification and self-awareness data. According to the architecture, the tag data coding protocols are defined at Tag Data Standards (TDS) and Tag Data Translation (TDT) which are extendable. The modification is made to the Serialized Global Transducer Item Number (SGXIN) [55] where the TDT file is appended in order to transform it to a *Thing* Data Translation. In this way, the new TDT can accommodate transducer and tag data concurrently. In the capture layer, the raw data acquired from the identify layer are filtered by the Application Level Events (ALE) middleware. The ALE middleware is an application of the corresponding standard and specifies the interface through which end users are able to obtain consolidated data and information about physical processes from a multitude of sources. The extended version of the ALE middleware is capable of handling raw data not only from the RFID tags but also from the smart transducers.

Finally, in the capture layer, the filtered and grouped raw data can act as input to capture service and generate specific events. The events are inserted into an extended EPC Information System (IS) which will be able to accept transducer functionalities. Then, the extended EPCIS stores the events and renders them available for query.

The integration of the IEEE1451 compatible transducers to the EPCglobal architecture is the main focus of the paper. Metadata are gathered from IEEE1451 TEDS structure and are appended to the identify layer of the EPCglobal standard (Fig. 17). The authors have distinguished as minimum meta-TEDS the transducer ID, sensing/triggering data, and other data required for handling all the relevant data as a block. These include the Universal Unique Identifier (UUID) which consists of metadata such as location (42 bits), manufacturer (4 bits) year (12 bits), and time (22 bits), summing up to a total of 80 bits.

The PnP nature of IEEE1451 is coupled with the well defined in RFID applications EPCglobal series of protocols to create a flexible framework for developing IoT produces/services.

Sensor Discovery and Configuration Framework for The Internet of Things Paradigm

Perera et al. [58] proposes another platform for automatic sensor discovery and configuration called *SmartLink*. The *SmartLink* architecture is based on a software entity that the authors have named as Context-Aware Dynamic Discovery of Things (CADDOT). The model consists of a total of eight phases that are performed either by the *SmartLink* or a cloud-based IoT middleware.

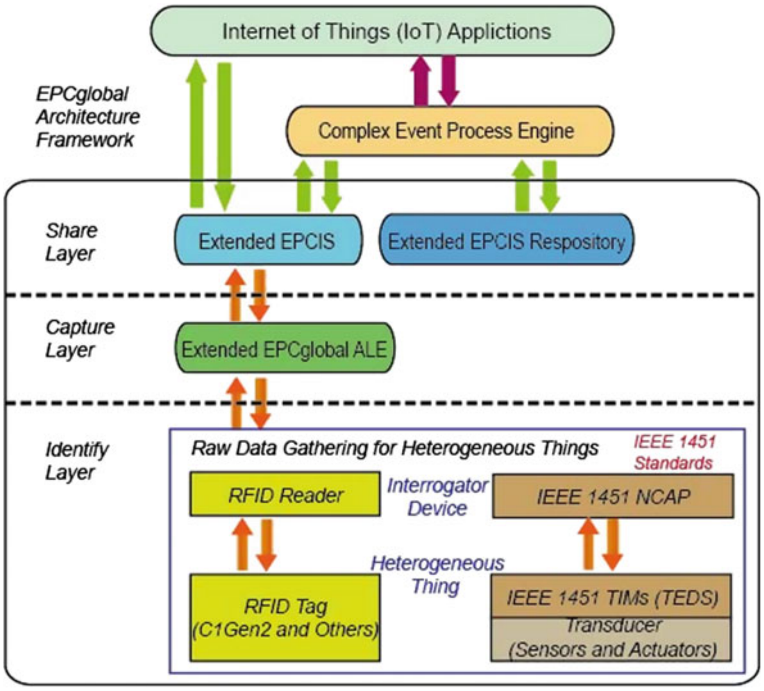

Internet of Things (IoT) Applictions
EPCglobal Architecture Framework
Complex Event Process Engine
Share Layer
Extended EPCIS
Extended EPCIS Respository
Capture Layer
Extended EPCglobal ALE
Identify Layer
Raw Data Gathering for Heterogeneous Things
IEEE 1451 Standards
RFID Reader
Interrogator Device
IEEE 1451 NCAP
Heterogeneous Thing
RFID Tag (C1Gen2 and Others)
IEEE 1451 TIMs (TEDS)
Transducer (Sensors and Actuators)

Fig. 17 EPCglobal extension with IEEE 1451 capability

The eight steps are:

- *Detect*: Here, the assumption is that the sensors are configured to seek for networks (Wi-Fi/Bluetooth) to connect to without the need of authorization. *SmartLink* is configured to act as an open wireless hotspot so that sensors connect to it in an ad-hoc fashion.
- *Extract*: In this phase to bypass the heterogeneous messages that are needed to identify every sensor, the authors propose to add an extra layer of communication information with which every sensor connected to the ad-hoc network of *SmartLink* will be able to respond to the message "WHO". The response to this message is the minimum amount of information for a sensor to be identified such as the unique identification number, model number/name, and manufacturer. After this response, the *SmartLink* has the required information to proceed with further interactions with the sensor through the sensor's native communication protocol. This approach of identification is similar to the TEDS algorithm mentioned earlier.
- *Identify*: During this phase, *SmartLink* sends response from the newly detected sensor to a cloud-based IoT middleware. The middleware queries its databases and retrieves every data available regarding the sensor. With this procedure, the sensor's profile in identified completely.

- *Find*: After the sensor module is fully identified, the IoT middleware pushed the necessary software drivers from the its database to the *SmartLink* where they are installed.
- *Retrieve*: At this point, *SmartLink* is capable of full communication with the sensor. Using this capability, the sensor is queried on any additional configuration details might contain such as schedules, sampling rates, data structures, etc. Further, *SmartLink*, if possible, it will communicate with other online sources to retrieve additional useful information related to the sensor.
- *Reason*: In this step, a context-aware sensing software is deployed. The IoT middleware takes into account the inputs from multiple sources to evaluate the capabilities, limitations, and operation details of every sensor. Following an optimization process a comprehensive sensing plan for each individual sensor is designed.
- *Configure*: In this last phase, sensors and cloud-based IoT software systems are going through final configurations. Schedules, communication, sampling frequency, and other details that were designed in the previous step are installed on the sensors. Communication between the sensors and the IoT cloud software is established through direct connection or through network capable gateways. Finally, communication configuration such as IP address, port, and authorization are provided by the *SmartLink*.

The authors point out possible application of this architecture to home automation and/or to agriculture IoT. A home automation system based using a Raspberry Pi as *SmartLink* was developed to demonstrate the effectiveness of the scheme.
Agile Manufacturing
Authors in [59] discuss about *Agile Manufacturing* [60] and how IoT in conjunction with PnP models can act as a strong enabler for industrial systems that will be easily configured and installed. Also, the notion of IoT@Work [61, 62] is mentioned, in which large number of intelligent devices are automatically configured in a similar manner to contemporary USB devices.

The discussion is about PnP systems and modules used in industrial premises. After thorough analysis of numerous manufacturing processes and systems, the IoT@Work project has been identified and is aiming to fulfill eight general requirements (A-Gx) towards PnP agile industrial systems:

- A-G1. System modification should be allowed during runtime, if the remaining systems in the production process are not affected.
- A-G2. When modifications require the production system to be in an offline state, rapid re-initialization should be possible.
- A-G3. When the system is modified during run-time, controlled initialization should be realized.
- A-G4. In case of communication faults, automatic network rerouting has to be supported.
- A-G5. Provision of a flexible and independent system bootstrapping.
- A-G6. Device migration and rapid device reconfiguration has to be supported.
- A-G7. Minimization of manual effort for configuration and initialization of the system.

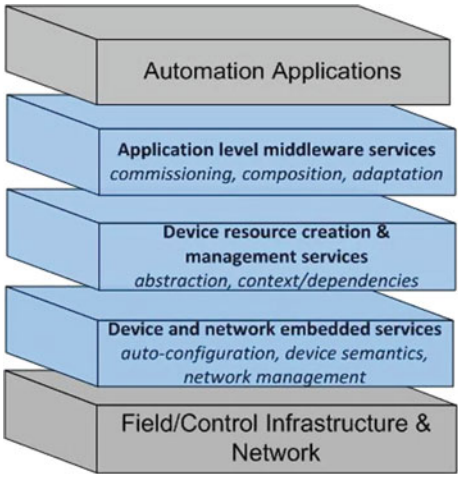


Fig. 18 IoT@Work layered architecture
Automation Applications
Application level middleware services
commissioning, composition, adaptation
Device resource creation & management services
abstraction, context/dependencies
Device and network embedded services
auto-configuration, device semantics, network management
Field/Control Infrastructure & Network

- A-G8. Intelligent system responses to various events such as faults.

Furthermore, the initiative has identified four additional requirements specific for automotive manufacturing systems.

- A-A1. Provision of a Graphical User Interface (GUI) for network configuration and initialization.
- A-A2. Provision of a GUI for the network maintenance system.
- A-A3. Provision of flexible and reliable semantic addressing scheme.
- A-A4. Capability of remote control and maintenance, in case of external maintenance contractors.

To address the aforementioned requirements, IoT@Work proposed a layered system architecture. Each layer corresponds to a different functional group and the abstraction of the layers begin with the low-level embedded devices and its end point is automation applications. In particular, three functional groups are identified Fig. 18 and can be defined as:

- The lowest abstract layer includes all the devices and network infrastructure along with their management functions. These functions are identifier assignment, device semantic and context collection, communication interfaces configuration, etc.
- The middle layer refers to all the services and functions that become available through the existing infrastructure and installed hardware systems. The abstraction level of these resources are higher since a lot of details from single devices are hidden.
- The top layer of abstraction refers to all the control schemes and scenarios that can be developed to service specific applications. At this layer, the interpretation of the application logic is performed during configuration time and runtime.

An Integrated Device and Service Discovery with UPnP and ONS to Facilitate the Composition of Smart Home Applications

Mitsugi et al.[63] proposes a complete PnP solution oriented towards smart home applications. Specifically, the authors present a protocol with *device* and *service* discovery capabilities by integrating the Universal Plug-and-Play protocol (UPnP) with the Object Naming Service (ONS) [64].

UPnP can keep a list of the available devices and services using its simple service discovery protocol. Every time that a new device joins the local network, it sends a message *ssdp:alive* to the gateways of the network in order to inform its existence. However, the gateway corresponds with the device using XML over HTTP. Such an implementation in many cases might not be possible because of the resource-constrained nature (computation, memory, and networking) of many devices used in a IoT smart home environment.

To bypass this issue, the Object Naming Service (ONS) is integrated to UPnP protocol. The ONS refers to the global service directory based on the EPC-GS1 standard (4.3). A control point (gateway) collects the device identifiers and then retrieves the service through ONS. An ONS client installed locally, can query the Root ONS with an EPC as key and find all services associated with the EPC. By this, services are available to the developer to create smart home applications.

The integration of the EPC protocol with UPnP is possible using CoAP. During an *ssdp:alive* message in a CoAP frame, the URI option is used to determine the device's EPC, which will be its unique identity. The operation of the developed protocol is shown in Fig. 19. To update its list of available service in the network, UPnP has a feature called *m-search*. The authors, again taking into account the resource-constrained devices on which the protocol will be implemented, developed a *transparent m-search* which has been shown to perform better [65] in such circumstances.

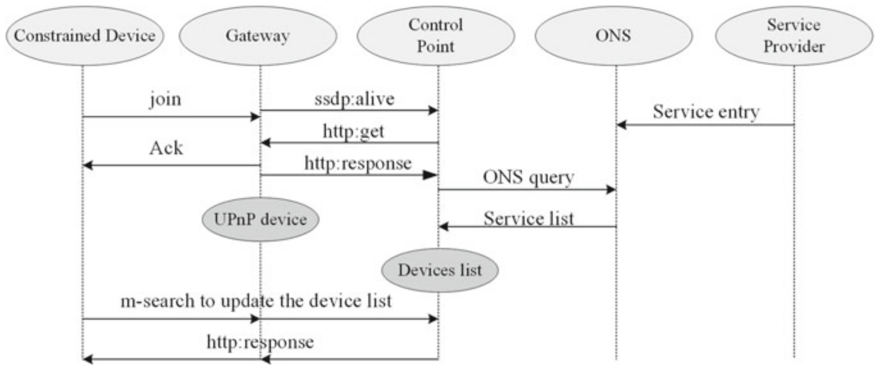


Fig. 19 A devices list is automatically generated in the control point. For constrained devices, a part of capability description may be obtained through ONS

5 Conclusion

In this chapter, an overview of the general PnP IoT architecture and a detailed description of the corresponding components was given. Furthermore, a wide survey was carried out and presented the most important models in the literature that feature PnP capabilities. It is evident that the field has demonstrated significant progress and has already showcased robust and complete PnP solutions. The main industrial players have focused their efforts to develop feature-rich ecosystems that present more and more components with PnP capabilities. However, the ecosystems support vertical architectures restricting the number of applications and IoT solutions that can be developed. In the following years, IoT manufacturers and companies are expected to perform a paradigm shift to interconnected horizontal systems that will benefit significantly the sector. Moreover, future development of PnP architectures will have to tackle challenges with increasing interest such as cybersecurity.

References

1. IEEE Standard for a Simple 32-Bit Backplane Bus: NuBus, ANSI/IEEE Std 1196-1987 (1988)
2. PCI-SIG. PCI Local Bus Specifications. https://pcisig.com/specifications. Accessed 15 Jan 2017
3. MSX Technical Data Book. Hardware/Software Specifications http://map.grauw.nl/resources/system/msxtech.pdf. Accessed 15 Jan 2017
4. IBM's Micro Channel Architecture. http://www.borrett.id.au/computing/art-1989-03-01.htm
5. Fred Krhenbhl: Micro Channel Architecture Bus Master Release 1.1: International Business Machines Corporation (1990)
6. ATmega328/P, 8-bit AVR Microcontrollers complete datasheet, Atmel. http://www.atmel.com/Images/Atmel-42735-8-bit-AVR-Microcontroller-ATmega328-328P_Datasheet.pdf
7. CC3200 SimpleLink Wi-Fi and Internet-of-Things Solution, a Single-Chip Wireless MCU, Texas Instruments. http://www.ti.com/lit/ds/swas032f/swas032f.pdf
8. SAM9G25, SMART ARM-based Embedded MPU datasheet, Atmel. http://www.atmel.com/images/atmel-11032-32-bit-arm926ej-s-microcontroller-sam9g25_datasheet.pdf
9. Raspberry Pi 3 Model B, Raspberry Pi Foundation. https://www.raspberrypi.org/products/raspberry-pi-3-model-b/
10. Adam Osborne, An Introduction to Microcomputers Volume 1: Basic Concepts, Osborne-McGraw Hill Berkeley California USA, 1980 ISBN 0-931988-34-9 pp. 116-126
11. UM10204, I^2C-bus specification and user manual, NXP Semiconductors, Rev.6 - 4 April 2014. http://cache.nxp.com/documents/user_manual/UM10204.pdf
12. How Plug and Play Works. https://technet.microsoft.com/en-us/library/cc781092. Accessed 15 Jan 2017
13. Apache-River Jini Architecture Specification. http://river.apache.org/release-doc/current/specs/html/jini-spec.html. Accessed 15 Jan 2017
14. The Java Virtual Machine Specification, Java SE 7 Edition, Tim Lindholm, Frank Yellin, Gilad Bracha, Alex Buckley, Oracle. Accessed 28 Dec 2013
15. Universal Plug and Play Device Architecture. http://upnp.org/specs/arch/UPnP-arch-DeviceArchitecture-v1.1.pdf. Accessed 15 Jan 2017
16. XML Protocol User Documents, XML Protocol Working Group. https://www.w3.org/2000/xp/Group/
17. User Datagram Protocol: RFC 768, IETF (1980). https://tools.ietf.org/html/rfc768

18. Service Location Protocol, Version 2. https://tools.ietf.org/html/rfc2608. Accessed 15 Jan 2017
19. Open Connectivity Foundation. https://openconnectivity.org/. Accessed 15 Jan 2017
20. IoTivity. https://www.iotivity.org/. Accessed 15 Jan 2017
21. Constrained Application Protocol Specification RFC7252. https://tools.ietf.org/html/rfc7252
22. Shelby, Z., Bormann, C.: 6LoWPAN: The Wireless Embedded Internet, vol. 43. Wiley (2011)
23. Yang, F., Matthys, N., Bachiller, R., Michiels, S., Joosen, W., Hughes, D.: PnP: plug and play peripherals for the internet of things. In: Proceedings of the Tenth European Conference on Computer Systems (EuroSys '15). ACM, New York, NY, USA, Article 25, p. 14 (2015). https://doi.org/10.1145/2741948.2741980
24. Matthys, N et al.: PnP-Mesh: The plug-and-play mesh network for the internet of things. In: IEEE 2nd World Forum on Internet of Things (WF-IoT), Milan, vol. 2015, pp. 311–315 (2015). https://doi.org/10.1109/WF-IoT.2015.7389072
25. Matthys, N., Yang, F., Daniels, W., Joosen, W., Hughes, D.: Demonstration of MicroPnP: the zero-configuration wireless sensing and actuation platform. In: 2016 13th Annual IEEE International Conference on Sensing, Communication, and Networking (SECON), London, pp. 1–2 (2016). https://doi.org/10.1109/SAHCN.2016.7732982
26. Yang, F., Ramachandran, G.S., Lawrence, P., Michiels, S., Joosen, W., Hughes, D.: PnP-WAN: Wide area plug and play sensing and actuation with LoRa. In: International SoC Design Conference (ISOCC), Jeju, vol. 2016, pp. 225–226 (2016). https://doi.org/10.1109/ISOCC.2016.7799869
27. Yang, F., Hughes, D., Joosen, W., Man, K.L.: The design of a low-power, high-resolution controller board for PnP. In: International SoC Design Conference (ISOCC), Gyungju, vol. 2015, pp. 173–174 (2015). https://doi.org/10.1109/ISOCC.2015.7401774
28. IEEE Standard for a smart transducer interface for sensors and actuators-common functions, communication protocols, and transducer electronic data sheet (TEDS) formats. In: IEEE Std 1451.0-2007, 21 Sept 2007, pp. 1–335. https://doi.org/10.1109/IEEESTD.2007.4338161
29. IEEE standard for a smart transducer interface for sensors and actuators-network capable application processor (NCAP) information model. In: IEEE Std 1451.1-1999, pp. i (2000) https://doi.org/10.1109/IEEESTD.2000.91313
30. IEEE standard for a smart transducer interface for sensors and actuators-transducer to microprocessor communication protocols and transducer electronic data sheet (TEDS) formats. In: IEEE Std 1451.2-1997, pp. i (1998). https://doi.org/10.1109/IEEESTD.1998.88285
31. IEEE standard for a smart transducer interface for sensors and actuators-digital communication and transducer electronic data sheet (TEDS) formats for distributed multidrop systems. In: IEEE Std 1451.3-2003, 31 Mar 2004, pp. 1–175. https://doi.org/10.1109/IEEESTD.2004.94443
32. IEEE standard for a smart transducer interface for sensors and actuators-mixed-mode communication protocols and transducer electronic data sheet (TEDS) formats. In: IEEE Std 1451.4-2004, pp. 01–430 (2004). https://doi.org/10.1109/IEEESTD.2004.95745
33. IEEE standard for a smart transducer interface for sensors and actuators wireless communication protocols and transducer electronic data sheet (TEDS) formats. In: IEEE Std 1451.5-2007 5 Oct 2007, p. C1-236. https://doi.org/10.1109/IEEESTD.2007.4346346
34. IEEE standard for smart transducer interface for sensors and actuators–transducers to radio frequency identification (RFID) systems communication protocols and transducer electronic data sheet formats. In: IEEE Std 1451.7-2010, 26 June 2010, pp. 1–99. https://doi.org/10.1109/IEEESTD.2010.5494713
35. Nieves, R., Madrid, N.M., Seepold, R., Larrauri, J.M., Arejita Larrinaga, B.: A UPnP service to control and manage IEEE 1451 transducers in control networks. IEEE Trans. Instrum. Meas. **61**(3), 791–800 (2012). https://doi.org/10.1109/TIM.2011.2170501
36. Depari, A., Ferrari, P., Flammini, A., Marioli, D., Taroni, A.: A VHDL model of a IEEE1451.2 smart sensor: characterization and applications. IEEE Sens. J. **7**(5), 619–626 (2007). https://doi.org/10.1109/JSEN.2007.894900
37. Kumar, A., Hancke, G.P.: A Zigbee-based animal health monitoring system. IEEE Sens. J. **15**(1), 610–617 (2015). https://doi.org/10.1109/JSEN.2014.2349073

38. Kumar, A., Hancke, G.P.: An energy-efficient smart comfort sensing system based on the IEEE 1451 standard for green buildings. IEEE Sens. J. **14**(12), 4245–4252 (2014). https://doi.org/10.1109/JSEN.2014.2356651
39. Chi, Q., Yan, H., Zhang, C., Pang, Z., Xu, L.D.: A reconfigurable smart sensor interface for industrial WSN in IoT environment. IEEE Trans. Ind. Inform. **10**(2), 1417–1425 (2014). https://doi.org/10.1109/TII.2014.2306798
40. Becari, W., Ramirez-Fernandez, F.J.: Electrogoniometer sensor with USB connectivity based on the IEEE1451 standard. In: IEEE International Symposium on Consumer Electronics (ISCE), Sao Paulo, vol. 2016, pp. 41–42 (2016). https://doi.org/10.1109/ISCE.2016.7797360
41. Bekan, A., Mohorcic, M., Cinkelj, J., Fortuna, C.: An architecture for fully reconfigurable plug-and-play wireless sensor network testbed. In: IEEE Global Communications Conference (GLOBECOM). San Diego, CA, vol. 2015, pp. 1–7 (2015). https://doi.org/10.1109/GLOCOM.2015.7417564
42. Fortuna, C., Mohorcic, M.: A framework for dynamic composition of communication services. ACM Trans. Sen. Netw. **11**(2), 32:132:43 (2014). https://doi.org/10.1145/2678216
43. Contiki: The Open Source OS for the Internet of Things. http://www.contiki-os.org/
44. RPL: IPv6 Routing Protocol for Low-Power and Lossy Networks, IETF. https://tools.ietf.org/html/rfc6550. Accessed Mar 2012
45. Mikhaylov, K., Huttunen, M.: Modular wireless sensor and Actuator Network Nodes with Plug-and-Play module connection. In: IEEE SENSORS: Proceedings Valencia **2014**, 470–473 (2014). https://doi.org/10.1109/ICSENS.2014.6985037
46. Mikhaylov, K., Paatelma, A.: Enabling modular plug-n-play wireless sensor and actuator network nodes: software architecture. In: IEEE SENSORS, Busan, vol. 2015, pp. 1–4 (2015). https://doi.org/10.1109/ICSENS.2015.7370252
47. Yang, F., Hughes, D., Matthys, N., Man, K.L.: The PnP Web Tag: A plug-and-play programming model for connecting IoT devices to the web of things. In: 2016 IEEE Asia Pacific Conference on Circuits and Systems (APCCAS), Jeju, South Korea, pp. 452–455 (2016). https://doi.org/10.1109/APCCAS.2016.7804000
48. Bordel, B., Rivera, D.S.D., Alcarria, R.: Plug-and-play transducers in cyber-physical systems for device-driven applications. In: 2016 10th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), Fukuoka, pp. 316–321 (2016). https://doi.org/10.1109/IMIS.2016.68
49. Dai, W., Huang, W., Vyatkin, V.: Enabling plug-and-play software components in industrial cyber-physical systems by adopting service-oriented architecture paradigm. In: IECON 2016-42nd Annual Conference of the IEEE Industrial Electronics Society, Florence, pp. 5253–5258 (2016). https://doi.org/10.1109/IECON.2016.7793834
50. IEC 61499: Function Blocks, International Standard, 2nd edn (2012)
51. Erl, T.: Service-Oriented Architecture: Concepts, Technology and Design, 760 pp. Prentice Hall Professional Technical Reference (2005)
52. Chan, S., Kaler, C., Kuehnel, T., Regnier, A., Roe, B. Sather, D., Schlimmer, J.: Devices Profile for Web Services. Microsoft Developers Network Library, Feb 2006
53. Khan, M.S., Islam, M.S., Deng, H.: Design of a reconfigurable rfid sensing tag as a generic sensing platform toward the future internet of things. IEEE Internet Things J. **1**(4), 300–310 (2014). https://doi.org/10.1109/JIOT.2014.2329189
54. EPC Radio-Frequency Identity Protocols Class-1 Generation-2 UHF RFID Conformance Requirements Version 1.0.5: GS1 EPCglobal. http://www.gs1.org/sites/default/files/docs/epc/uhfc1g2_1_1_0-Conformance%20Test%20Methods_1_0_5-20070323.pdf. Accessed Mar 2007
55. EPCTM 9 Generation 1 Tag Data Standards Version 1.1 Rev.1.27: GS1 EPC Global. http://www.gs1.org/sites/default/files/docs/epc/tds_1_1_rev_1_27-standard-20050510.pdf. Accessed May 2005
56. Cirani, S., et al.: A scalable and self-configuring architecture for service discovery in the internet of things. IEEE Internet Things J. **1**(5), 508–521 (2014). https://doi.org/10.1109/JIOT.2014.2358296

57. Tseng, C.W., Chen, Y.C., Huang, C.H.: Integrating transducer capability to GS1 EPCglobal identify layer for IoT applications. IEEE Sens. J. **15**(10), 5404–5415 (2015). https://doi.org/10.1109/JSEN.2015.2438074
58. Perera, C., Jayaraman, P.P., Zaslavsky, A., Georgakopoulos, D., Christen, P.: Sensor discovery and configuration framework for the internet of things paradigm. In: IEEE World Forum on Internet of Things (WF-IoT), Seoul vol. 2014, pp. 94–99 (2014). https://doi.org/10.1109/WF-IoT.2014.6803127
59. Houyou, A.M., Huth, H.P., Kloukinas, C., Trsek, H., Rotondi, D.: Agile manufacturing: general challenges and an IoTWork perspective. In: Proceedings of 2012 IEEE 17th International Conference on Emerging Technologies & Factory Automation (ETFA 2012), Krakow, pp. 1–7 (2012). https://doi.org/10.1109/ETFA.2012.6489653
60. Dai, L., Liu, L., Sun, X.: The system construction and research review of agile manufacturing. In: 2011 International Conference on Management and Service Science, Wuhan, pp. 1–4 (2011). https://doi.org/10.1109/ICMSS.2011.5998319
61. Rotondi, D., et. al.: D1.1 State of the art and functional requirements in manufacturing and automation; IoT@Work public deliverable. https://www.iot-at-work.eu/downloads.html. Accessed 30 Dec 2010
62. Houyou, A., Huth, H.-P.: Internet of things at work European project: enabling plug&work in automation networks. In: Embedded World Conference 2011, Nuremberg, Germany (2011)
63. Mitsugi, J., Sato, Y., Ozawa, M., Suzuki, S.: An integrated device and service discovery with UPnP and ONS to facilitate the composition of smart home applications. In: IEEE World Forum on Internet of Things (WF-IoT), Seoul vol. 2014, pp. 400–404 (2014). https://doi.org/10.1109/WF-IoT.2014.6803199
64. EPCglobal Object Name Service (ONS) 1.0.1 (2008)
65. Mitsugi, J., Yonemura, S., Yokoishi, T.: Reliable and swift device discovery in consolidated IP and ZigBee home networks. IEICE Trans B **E96-B**(07) (2013)