

Taking Arduino to the Internet of Things: the ASIP programming model

Gianluca Barbon, Michael Margolis, Filippo Palumbo, Franco Raimondi, Nick Weldin

▶ **To cite this version:**

Gianluca Barbon, Michael Margolis, Filippo Palumbo, Franco Raimondi, Nick Weldin. Taking Arduino to the Internet of Things: the ASIP programming model. Computer Communications, 2016, 00, pp.1 - 15. 10.1016/j.comcom.2016.03.016 . hal-01416490

HAL Id: hal-01416490
https://inria.hal.science/hal-01416490

Submitted on 14 Dec 2016

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Available online at www.sciencedirect.com
ScienceDirect
Computer Communications 00 (2016) 1–15
Computer Communi-cations

Taking Arduino to the Internet of Things: the ASIP programming model

Gianluca Barbon[a], Michael Margolis[b], Filippo Palumbo[c,d], Franco Raimondi[b], Nick Weldin[b]

[a] *University of Grenoble Alpes, Inria, LIG, CNRS, France*
[b] *Department of Computer Science, Middlesex University, London, United Kingdom*
[c] *Institute of Information Science and Technologies, National Research Council, Pisa, Italy*
[d] *Department of Computer Science, University of Pisa, Pisa, Italy*

Abstract

Micro-controllers such as Arduino are widely used by all kinds of makers worldwide. Popularity has been driven by Arduino's simplicity of use and the large number of sensors and libraries available to extend the basic capabilities of these controllers. The last decade has witnessed a surge of software engineering solutions for "the Internet of Things", but in several cases these solutions require computational resources that are more advanced than simple, resource-limited micro-controllers.

Surprisingly, in spite of being the basic ingredients of complex hardware-software systems, there does not seem to be a simple and flexible way to (1) extend the basic capabilities of micro-controllers, and (2) to coordinate inter-connected micro-controllers in "the Internet of Things". Indeed, new capabilities are added on a per-application basis and interactions are mainly limited to bespoke, point-to-point protocols that target the hardware I/O rather than the services provided by this hardware.

In this paper we present the Arduino Service Interface Programming (ASIP) model, a new model that addresses the issues above by (1) providing a "Service" abstraction to easily add new capabilities to micro-controllers, and (2) providing support for networked boards using a range of strategies, including socket connections, bridging devices, MQTT-based publish-subscribe messaging, discovery services, etc. We provide an open-source implementation of the code running on Arduino boards and client libraries in Java, Python, Racket and Erlang. We show how ASIP enables the rapid development of non-trivial applications (coordination of input/output on distributed boards and implementation of a line-following algorithm for a remote robot) and we assess the performance of ASIP in several ways, both quantitative and qualitative.

Keywords: Arduino, MQTT, IoT, Service Discovery, Communication Middleware

1. Introduction

The Internet of Things (IoT) paradigm bases its success on the pervasive presence around us of a variety of objects (such as Radio-Frequency IDentification -RFID- tags, sensors, actuators, mobile phones, etc.) which, through unique addressing schemes, are able to interact with each other and cooperate to reach common goals [1].

Surprisingly, in spite of being the basic ingredients of complex hardware-software systems, there does not seem to be a simple and flexible way to (1) extend the basic capabilities of micro-controllers, and (2) to coordinate inter-connected micro-controllers in IoT scenario. Indeed, new capabilities are added on a per-application basis and interactions are mainly limited to bespoke, point-to-point protocols that target the hardware I/O rather than the services provided by this hardware.

Several commercial off-the-shelf devices are available on the market, but usually they are tightly coupled with specific vendors and require local gateways to export sensors and actuators as services on the Web. Instead, by embracing the open source and hardware principles, it is possible to offer a system easily modifiable to suit the user needs and to be used as the basis for new products in different scenarios. Micro-controllers such as Arduino are used widely by all kinds of makers worldwide. Popularity has been driven by Arduino's simplicity of use and the large number of sensors and libraries available to extend the basic capabilities of these controllers. Using such an inexpensive device makes the installation and maintenance of a system easier. In this way, it is possible to offer a system easily modifiable to suit the user needs and to be used as the basis for new products in different scenarios.

Email addresses: gianluca.barbon@inria.fr (Gianluca Barbon), m.margolis@mdx.ac.uk (Michael Margolis), filippo.palumbo@isti.cnr.it (Filippo Palumbo), f.raimondi@mdx.ac.uk (Franco Raimondi), n.weldin@mdx.ac.uk (Nick Weldin)

A first approach using the Internet for interacting with real-world resource-constrained devices was to incorporate smart things into standardized Web service architectures (such as SOAP, WSDL, UDDI) [2] or embedding HTTP servers into the devices. However, in practice, this resulted to be too heavy and complex for simple objects [3]. In order to face the problem of interconnecting several resource-constrained nodes among each other and to the Internet, several communication protocols have been introduced [4]. These protocols are inspired by machine-to-machine (M2M) scenarios and share the same fundamentals of communication paradigms typical of standard computer networks. M2M communications occur among machines (objects or devices) with computing/communication capabilities without human intervention [5].

In this paper we present the Arduino Service Interface Programming (ASIP) model, a new model that addresses the issues above by (1) providing a "Service" abstraction to easily add new capabilities to micro-controllers, and (2) providing support for networked boards using a range of strategies, including socket connections, bridging devices, MQTT-based publish-subscribe messaging, discovery services, etc. We provide an open-source implementation of the code running on Arduino boards and client libraries in Java, Python, Racket and Erlang. Our programming model allows to tackle the heterogeneity that is a distinguishing feature of several IoT applications; by heterogeneity we mean here hardware differences (different microcontrollers), performances/capabilities of different boards in terms of CPU power, memory, and storage, and software heterogeneity (e.g., choice of programming languages). We show how ASIP enables the rapid development of non-trivial applications (coordination of input/output on distributed boards and implementation of a line-following algorithm for a remote robot) and we assess the performance of ASIP in several ways, both quantitative and qualitative.

The rest of the paper is organized as follows. We review related work in Section 2; we introduce the ASIP model in Section 3, describing the software architecture, the communication protocol and the possible communication channels: serial, TCP, and MQTT publish/subscribe messaging. We present a detailed experimental evaluation of ASIP performance in Section 4. In Section 5, instead, we give a qualitative evaluation by providing examples of how applications can be built on top of ASIP.

2. Related Work

The emerging IoT scenario, exploiting the advances made in the M2M field, enables the possibility of building a huge Service Oriented Architecture (SOA) composed of several devices offering services each other [2]. Existing application platforms use REST architecture [6, 7, 8, 9, 10, 11, 12, 13, 14] as interfaces in order to expose their services. The REST-style architecture consist of clients and servers. Clients initiate requests to servers; servers process requests and return the appropriate responses manipulating the resources. A resource can be any thing identified by URIs. REST uses the GET, PUT, POST, and DELETE operations of HTTP to access resources. However, the protocols used for RESTful architecture are not appropriate for resource constrained networks and devices [15]. The large overhead of HTTP causes packet fragmentation and performance degradation when dealing with M2M devices. Also, TCP flow control is not appropriate for resource-constrained devices and the overhead is too high for short transactions.

To extend the REST architecture for resource-constrained devices, a first solution presented in the literature is given by the Constrained Application Protocol (CoAP) [16]. CoAP is a protocol intended to be used in simple devices allowing them to communicate over the Internet. CoAP includes a subset of the HTTP functionalities, optimized for M2M applications. It also supports multicast, very low overhead, and asynchronous message exchanges over a user datagram protocol (UDP) [17]. However, also in CoAP, the HTTP protocol is still present. It has not been designed to support persistent communication and, even if Web Sockets have been introduced in the recent draft of HTML 5 (offering a bidirectional communication channel between client and server), they totally hide the naming scheme that makes REST so powerful: every resource having a standard unique identifier, the URI. The Web Sockets approach results in non-standard solutions for manipulating resources [18]. In order to support collaboration between devices, there is the need to unify the naming scheme of smart objects and the URIs. For this purpose, a new communication paradigm has been presented in the literature based on the "publish-subscribe" (pub/-sub) mechanism [19, 20].

Directed Diffusion [21] is considered the earliest pub/sub communication paradigm for WSN. It is a data-centric protocol in which named data is described by attribute-value pairs. The subscriptions are called interests and are broadcasted throughout the whole WSN. Another early pub/sub middleware for WSN is Mires [22]. It is implemented on top of TinyOS [23], an event-based operating system for WSNs. In Mires each sensor advertises its topics (e.g. temperature, pressure, luminosity, etc.) to the applications through a sink node. A slightly different programming approach is used by TinyCOPS [24], a component-based middleware that also provides a content-based pub/sub service to WSN that tries to simplify the composition of services through components (communication protocol, supported data, and service extensions). Recently, Object Management Group (OMG) has published DDS, an open standard for data-centric publish-subscribe middleware platforms with real-time capabilities [25]. TinyDDS [26] is the adopted version of OMG DDS for WSN, based on TinyOS. It is a lightweight pub/sub middleware that allows applications to interoperate across the boundary of WSNs and access networks, regardless of their programming languages and protocols.

All of these solutions are strictly coupled with the hardware platform and they usually need centralized hardware infrastructure (sink nodes or gateways) [27, 28]. Also the MQTT protocol introduced by IBM uses hierarchical topic based [29] publish-subscribe mechanism and facilitates the constrained devices by enabling "pushing" [30] data from the cloud rather than polling by constrained device for the data from the server. In this case the overlay infrastructure is a software component, the broker. The broker is responsible for distributing messages to interested

clients based on the topic of a message. In the sensor domain, IBM has come up with yet another protocol MQTT for Sensor Networks (MQTT-SN) [31] which is designed in such a way that the protocol is agnostic of the underlying networking services.

From the devices availability point of view, there are two types of hardware platforms that can be connected to IoT service platforms. One is off-the-shelf commercial products that are related to certain platforms, for example, Cosm consumer products [6], ioBridge [32], NanoRouter [17], MicroStrain Sensors [33], and Digi routers [14]. The second type is an open hardware (development/hackable) platform that users can develop themselves, such as Arduino [34], mBed [35], or Nanode [36]. Embracing the open source and hardware principles, it is possible to offer a system easily modifiable to suit the user needs and to be used as the basis for new products in different scenarios.

Different solutions have been proposed in order to bring the IoT paradigm on the Arduino platform. In the healthcare field, a textile version of the Arduino platform, called LilyPad [37], has been used to bridge wearable medical devices to IoT enabled infrastructure using a mobile device as gateway. In [38], authors propose a model to enable the event reading and the controlling of electrical devices using a master controller that acts as a gateway that is a standard PC. In the energy monitoring scenario, [39] describes a non-intrusive load monitoring system for domestic appliances where a web server is embedded on the Arduino board. These solutions embed HTTP web servers on board that make easier to fetch the exposed information in one-to-one client-server connections. Our aim is to avoid the presence of local gateways or embedded resource-consuming web servers, offering the possibility to coordinate inter-connected micro-controllers through Internet providing support for networked boards with different strategies: socket connections, bridging devices, and MQTT-based pub/sub messaging.

Pairwise evaluations and comparisons of HTTP, CoAP and MQTT protocols have been reported in the literature. For example, [40, 41, 42] compares the performance of MQTT, CoAP, and HTTP in terms of end-to-end transmission delay and bandwidth usage [41] and in terms of energy consumption and response time [42]. Based on their results, MQTT delivers messages with lower delay than CoAP when the packet loss rate is low, while, due to its condensed header and small packet size, CoAP is more efficient than HTPP in transmission time and energy usage. Regarding the power consumption of the devices, there is a detailed experiment for power consumption comparison between HTTP and MQTT on mobile devices [43]. The result shows that the MQTT protocol wins in all tests, which include establishing, maintaining, and receiving/sending messages [44]. From these considerations, we choose to implement a simple text-based service discovery mechanism to let several resource-constrained microcontrollers to discover each other with their functionalities and to exchange messages among them.

From the programming model point-of-view, in order to expose sensors and actuators as services, the components connected to the micro-controller need to be programmed individually to take into account both low-level implementation details and the high-level requirements of the application of which the micro-controller is part. Several solutions have been proposed in order to address the issue of abstracting low-level implementation details [45]. Most of them are tightly targeted to particular applications [46] or hardware platforms [23]. Instead of relying on a dedicated operating system, in this paper we propose that sensors and actuators are exposed by micro-controllers as *services*, so that more complex software applications can be built by composing them. When restricted to the Arduino platform, the closest protocol to ASIP is the Firmata protocol [47], which enables a computer to discover, configure, read and write a microcontrollers general purpose IO pins. However, ASIP has a smaller footprint than Firmata (using around 20% less RAM). And uniquely, it supports high level abstractions that can be easily attached to hundreds of different services for accessing sensors or controlling actuators. These abstractions can decouple references to specific hardware, thus enabling different micro-controllers to be used without software modification. Although ASIP is currently implemented for Arduino boards, the protocol is hardware agnostic. Moreover, as shown below, ASIP supports communication over TCP and MQTT, while Firmata is limited to serial communication.

3. The ASIP programming model

In this section we describe the Arduino Service Interface Programming model (ASIP), which has been developed to simplify and accelerate the development of applications in the IoT. Applications for the IoT typically involve heterogeneous components, both in terms of software and hardware. *Machine-to-machine* communications are the prevalent mechanism for coordination and execution of tasks. The ASIP programming model addresses this issue, together with a mechanism to integrate with existing protocols such as MQTT. We note that, while providing a seemingly Arduino-specific solution, our programming model is generic and can be implemented on top of microcontrollers with very limited resources. Our choice for Arduino has been motivated by the open-source nature of the project and by the availability of hardware. Specifically, ASIP builds upon the notion of "service" for micro-controllers: a service could be a temperature sensor, a servo motor, or any other input or output device connected to a micro-controller. Each micro-controller can be controlled using textual messages, and each micro-controller reports data using messages to so-called *clients*. The core ASIP implementation running on a micro-controller deploys one or more services: this enables the reusability of both micro-controller-specific code and of client code, and it opens the possibility of model-based development for complex applications involving multiple micro-controllers, as described below.

In summary, the ASIP model provides:

- A software architecture for code running on the micro-controller.

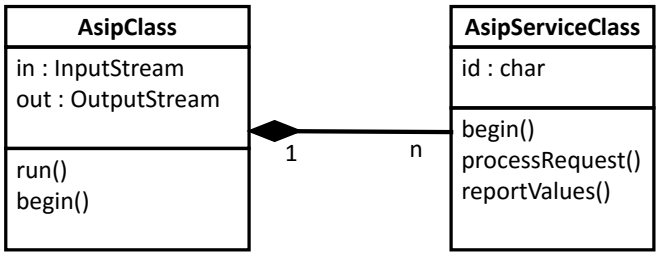


Figure 1: ASIP simplified class diagram

- A textual protocol for messages exchanged between ASIP clients and micro-controllers implementing the software architecture mentioned above.
- A network architecture for the connection between micro-controllers and client that can be written in several high-level programming languages.

These various components are described in the subsections below.

3.1. Basics: software architecture

As mentioned above, at the core of ASIP is the notion of service. We model it by means of the class `AsipServiceClass` (see right-hand side of Figure 1). Each service, e.g. a distance sensor, must have a unique ID and it can reuse existing Arduino libraries developed specifically for the given component (sensor, shield, motor, etc.) to obtain data from that component. Each service must implement the following methods:

- `begin()`, to set up the service appropriately, for instance by initialising the pins or by enabling interrupts.
- `processRequest()`, to process messages for the service dispatched by the class `AsipClass`, as described below.

If the service returns values, for instance in the case of a temperature or distance sensors, then the service should also implement the method `reportValues()`. This method converts data into ASIP messages, using the syntax of ASIP messages described below. A number of services is already provided with the ASIP implementation that we describe in this paper, but additional ones could be defined by implementing an AsipServiceClass to handle appropriate messages. It is assumed that all implementations of ASIP support at least the basic Input/Output ASIP service, which provides basic I/O operations at the pin level. On an Arduino board these operations include writing and reading values from digital and analog pins, thus permitting the control of LEDs or reading potentiometer values.

Service are put togethet in the class `AsipClass` (left-hand-side of Figure 1. The AsipClass is the core of ASIP and is responsible for managing services. The AsipClass on the microcontroller is connected to a stream, which can be a serial channel, a TCP socket or a MQTT pub/sub mechanism (please refer to the system architecture described below for additional details). The `AsipClass` must implement a run method that executes the main ASIP loop. Before the execution of the main loop, an initialization mechanism is called to set up the communication streams. The main loop performs the core operations to handle ASIP, acting like a dispatcher of messages to/from services. First of all, it listens for incoming messages, and redirects them to the proper service by recognizing the service identifier in the ASIP message header. A particular set of messages, called systems messages, are not handled though a service but are processed through proper methods supplied by the AsipClass. Moreover, the loop allows services to reply continuously in case periodic status messages have been enabled, for instance to report a distance reading at regular time intervals.

3.2. The syntax of ASIP messages

Messages exchanged between micro-controllers and clients are plain text messages with a standard format. They can be divided into *command* messages and *event* messages. The first are sent by ASIP clients to micro-controllers, while the latter are sent on the opposite direction by the micro-controller. ASIP messages consist of an ASCII header, followed by ASCII character fields separated by commas, and terminated by the newline character.

Command (or request) messages *to* a micro-controller begin with a single character to indicate the desired service, followed by a comma and a single character tag to identify the nature of the request. Requests that contain a parameter are separated from the tag with a comma. As an example, the message `I,d,13,1` invokes the service with ID `I` (typically, an Input/Output service), requesting an operation `d` (in this case it is a request to write on a digital pin) with parameters 13 and 1. These parameters indicate, respectively, pin 13 and the value 1 (high).

In the other direction, reply messages *from* the microcontroller begin with one of the following characters:

- "@" defines an event message responding to a request or autoevent. These messages are composed of three bytes following the "@" character: a character indicating the service, a comma, and the tag indicating the request that triggered this event respectively.
- "~" defines an error message reporting an ill formed request or some other problems affecting the server. These messages contain the service and tag associated with the error followed by an error number and error string.
- "!" defines an informational or debug message consisting of unformatted ASCII text terminated by the newline character.

Some reply event messages have a payload with a variable number of fields with the following format:

- a numeric value that precedes the message body indicating the number of fields in the body

Request explicit distance measurement:

Header	Separator	Tag	Terminator
'D'	,	'M'	'\n'

Request distance autoevents:

Header	Separator	Tag	Separator	Period	Terminator
'D'	,	'A'	,	Numeric value in milliseconds	'\n'

Reply:

Header	Separator	Tag	Separator	Distance in CM	Terminator
'D'	,	'M'	,	Numeric digits	'\n'

Figure 2: ASIP Messages: example of syntax for a distance service

- curly brackets used to indicate the start and end of fields in the message body
- if a message contains sub fields, these are separated by a colon (for instance, in case of analog pin mapping message)
- all numeric values are expressed as ASCII text digits and are decimal unless otherwise stated.

Figure 2 reports the syntax for a distance service.

3.3. System Architecture

Micro-controllers can be connected to clients in a number of way: directly using a serial connection (over USB), by means of TCP sockets, or using an MQTT-based publish/subscribe messaging mechanism.

Serial sockets and TCP connections are used in point-to-point connections, when a client has exclusive access to a device, for instance for controlling a robot. The MQTT-based architecture allows sending and receiving data to and from multiple devices, thus resulting useful in applications such as sensor networks (smart homes, etc.).

3.3.1. Serial connection

The serial connection uses the USB bus in order to connect to a micro-controller from a computer. This is the basic configuration for the ASIP architecture. The micro-controller must run an implementation of the `AsipClass` described above, with at least the implementation of the Input/Output Service. Theoretically, on the computer side of the connection, a simple serial monitor could send instructions and read values being reported. In practice, applications are written in a high-level language to make use of the services installed on the micro-controller. Various implementations of the serial client exist for different languages:

- Java:
  `https://github.com/fraimondi/java-asip`
- Python:
  `https://github.com/gbarbon/python-asip`
- Racket:
  `https://github.com/fraimondi/racket-asip`
- Erlang:
  `https://github.com/ngorogiannis/erlang-asip`

3.3.2. TCP and MQTT bridges

Before introducing the TCP and the MQTT architecture we introduce the notion of *bridge* to address the issue of network connection for micro-controllers. Indeed, a micro-controller such as an Arduino board needs an additional ethernet or a wi-fi device to communicate on a network. This device may be an Arduino shield, or an external device. In the first case, the Arduino ASIP client sketch integrates the code needed to talk to the shield. In the latter case, instead, the Arduino sketch can be connected using a standard serial connection to the external device and it does not require modifications. Thus, it is the external device that will take care of the network communications. We call this kind of devices *bridges*.

The bridge logic is very simple: the bridge listens to incoming messages from the network and routes them to the serial port. In the other direction, the bridge listens to incoming messages from the micro-controller over its serial connection and it redirects them to the network interface. A bridge does not implement ASIP classes or services, because its only function is to permit the communication between different transmission medium. In order to avoid the presence of errors in the conversion between two transmission channels, error checking can be implemented.

Bridges can be implemented using different kinds of hardware platforms. For the testing phase of this paper, the Raspberry PI 2 has been used as hardware platform. However, lighter platforms can be adopted [48] and we have successfully employed the ESP8266 chip, a lightweight SoC that features low power consumption and includes a wi-fi antenna [1].

Bridges can be employed both for TCP and for MQTT connections, as explained below. The Java and Python implementations of ASIP clients include bridges for TCP and for MQTT.

[1] See `https://github.com/michaelmargolis/mdx_prototypes/tree/51145bd1347625df9bc4e79b7eb11726f243a401/ASIP/Lua`.

The adoption of bridges brings some advantages:

- a single board can be used by different clients;
- use of low cost devices instead of expensive Arduino shields (like the Ethernet and Wi-Fi shields);
- use of devices with very low power consumption;
- reduction of the workload on the Arduino board.

3.3.3. TCP

Connecting to an Arduino through TCP may require the adoption of a bridge, depending on how the micro-controller is connected to the network. A micro-controller implementing ASIP messaging over TCP must have an IP address and a dedicated TCP port open to connections. Once a client opens a connection to the address and port creating a socket, the exchange of messages continues identically to the serial communication. In fact, client applications developed to work over serial communication can be immediately translated into TCP-based applications just by replacing the client connection class. Code for TCP clients is provided in the Java and Python repositories mentioned above.

Even if the micro-controller is limited to single connections, nothing prevents the client from opening connections to multiple, networked devices, thus enabling the coordination of networked micro-controllers. Notice that sockets can be created even over internet, thus enabling the control of possibly very remote micro-controllers. In addition, if an appropriate DNS record can be provided for each micro-controller, standard domain naming mechanisms can be used to identify boards and bridges.

3.3.4. MQTT

ASIP messages can be exchanged using MQTT. The main advantage is in the implementation of clients that need to connect to multiple boards: instead of opening a socket for each micro-controller, the client can simply subscribe and publish messages to a broker. Similarly, MQTT bridges forward serial messages from the board to publish actions for appropriate topics and route subscribed messages to the serial channel. More in detail, we assume that each micro-controller is identified by a unique name and we employ the following scheme for topics:

- Messages *from* a specific board are published by the board (or by its bridge) to the MQTT broker with topic `asip/BOARDNAME/out`, where `BOARDNAME` is the unique identifier of the micro-controller.
- Messages *to* a specific board are published by clients to the MQTT broker with topic `asip/BOARDNAME/in`.

In practical terms, an MQTT bridge for a board should *subscribe* to topic `asip/BOARDNAME/in` and forward messages received on this topic to the serial connection. The same bridge should *publish* serial messages from the board to topic `asip/BOARDNAME/out`. Conversely, a client of a specific board should subscribe to topic `asip/BOARDNAME/out` to receive messages from the board, and publish messages to `asip/BOARDNAME/in` to send messages to the board.

Implementations of MQTT bridges and sample clients are available in the Java and Python repositories.

3.4. Service-level discovery mechanism

In order to let the devices know the presence of other devices and their embedded sensors/actuators exposed as services, we propose a service discovery overlay that exploits the MQTT protocol capabilities. We follow the approach presented in [49], where the service discovery functionality is realized by intelligent buses, namely the *context*, *service* and *control* buses. All communications between devices can happen in a round-about way, via one of them. Each of the buses handles a specific type of message/request and is realized by different kinds of MQTT topics. This approach has been proven useful in different application scenarios, spanning from ambient assisted living [50], mobile [51], and energy monitoring [52] infrastructures.

In particular, as soon as a micro-controller is turned on, it announces his presence publishing a message containing his unique identifier on the service bus topic (`asip/servicebus`) and it subscribes to the same topic in order to be notified by the MQTT broker about existing (already announced) or new services to be announced. In this way, all the devices can discover its presence and start to listen for its messages subscribing to the relative context bus topic (`asip/BOARDNAME/out`). Micro-controllers, if capable, can also accept commands, subscribing to their control bus (`asip/BOARDNAME/in`) and waiting for messages published by other micro-controllers/services on the same topic. Figure 3 shows an example of a possible interaction among different micro-controllers (`B1,...,B4`) and a remote service translating MQTT topics to REST resources (the description of this kind of service is out of the scope of this paper, see [18, 44] for possible solutions). A micro-controller can also expose different sensors/actuators as services (`B4` in the figure announcing sensor `S1` and actuator `S2`). In the example, `B4` announces itself and its sensors/actuators publishing the relative identifiers on the service bus topics:

```
PUBLISH asip/servicebus B4
PUBLISH asip/servicebus/B4 S1
PUBLISH asip/servicebus/B4 S2
```

then, it starts publishing data from `B4/S1` on the relative context bus topic and it subscribes to the context bus topic relative to `B4/S2`, waiting for incoming commands:

```
PUBLISH asip/B4/S1/out data
SUBSCRIBE asip/B4/S2/in
```

In the depicted example, micro-controllers `B1`, `B2`, and `B3` are consumers of `B4/S1` and they have already subscribed to the relative context bus topic, so they start receiving the required data. In the meantime, the REST service exposes `B4/S1` and `B4/S2` as web resources and can send a command to the actuator `B4/S2` publishing on the relative control bus topic:

```
PUBLISH asip/B4/S2/in command
```

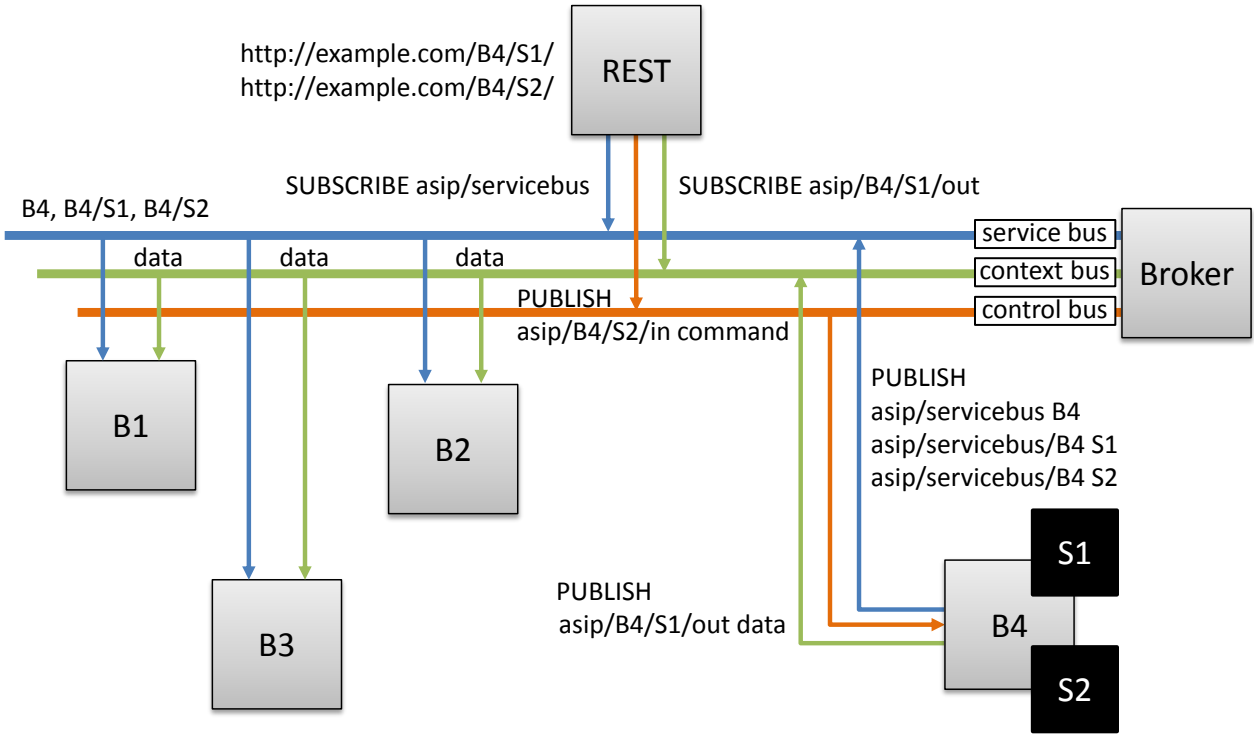


Figure 3: An application scenario exploiting the service discovery functionality.

4. Quantitative Evaluation

In this section we provide a quantitative experimental evaluation of ASIP. In particular, we to assess the *throughput* (how many messages per second can be sent?) and the *latency* (what is the delay between a request and a response?) for the possible network architectures described above: direct serial connection, TCP socket, and MQTT publish/subscribe.

4.1. Throughput

In this work we define throughput for ASIP as the number of messages per second that can be sent over a communication channel. Intuitively, this corresponds to the maximum frequency of updates that can be achieved, for instance to control a robot.

4.1.1. Experimental set-up

Figure 4 sketches our experimental set-up. At a high level, we use a signal generator to generate periodic impulses that are received by a micro-controller running ASIP on input pin 2. A client is connected to the Arduino using one of the possible channels (serial, TCP, MQTT) and it sets the value of output pin 13 according to the value read on pin 2. We then use an external oscilloscope to track the original signal entering pin 2 and the signal generated by the ASIP client on pin 13 to make sure that the frequencies are the same. If this is the case, then ASIP can process this number of messages per second. More in detail, in Figure 4:

- The Arduino depicted on top acts as the signal generator by emitting a signal on pin 9. The Arduino is running a simple sketch that generates a periodic signal with a specific frequency (in our sketch this value can be changed on-the-fly).
- The Arduino depicted on the lower part of the figure runs ASIP and is connected to a client through pins 0 and 1.
- The Raspberry Pi runs ASIP client code, which could be written in Java or in Python. Notice that the Raspberry Pi could be replaced by a laptop connected to the Arduino using a USB connection, or it could be replaced by a bridge for TCP or MQTT, which could in itself be a

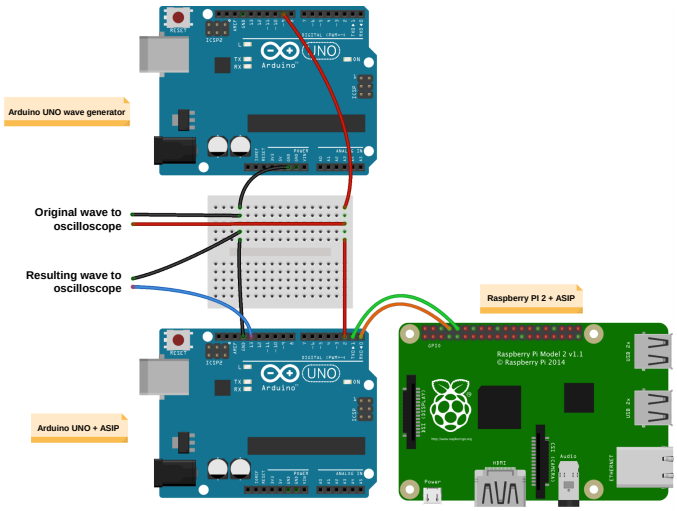


Figure 4: Serial testing set-up.

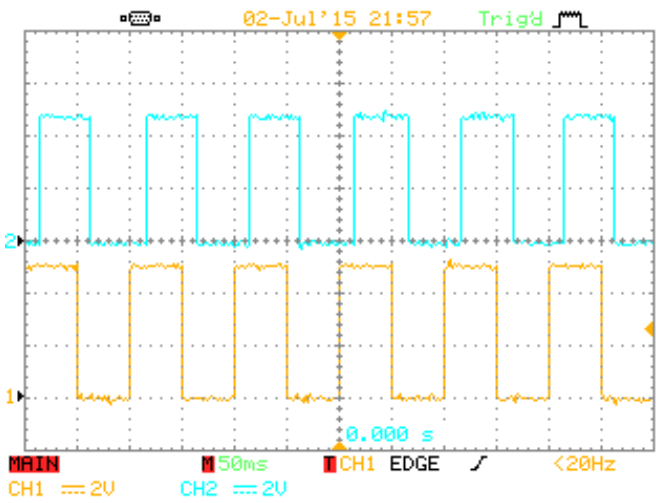


Figure 5: Oscilloscope output.

Raspberry Pi or an ESP 8266 chip.

- An external two-input oscilloscope (not depicted in the figure) compares the signal generated by the Arduino depicted on top with the signal generated on pin 13 of the Arduino depicted on the bottom.

We used an oscilloscope in order to see the difference between the wave generated by the wave generator and the resulting wave obtained after "travelling" through the ASIP network. This allowed to have a visual feedback about the throughput, in order to check the maximum rate allowed by ASIP. An example output for the oscilloscope is depicted in Figure 5. In this figure, the signal in the lower part is the signal from the signal generator, while the signal on top is the signal from the ASIP board (the drift between the two waves is the latency, assessed separately in the section below). The figure depicts a frequency for which the ASIP client can track the signal correctly, because the number of peaks is the same in both traces. When the frequency of the original signal increases above a certain threshold, the top wave fails to track the signal and misses some of the peaks.

Notice that the logic of the test is all performed in the ASIP client. The incoming signal from pin 2 is sent through an ASIP message from Arduino to the client. The client processes the message, reads the value and establishes the value of pin 13. Finally it sends a message to the Arduino with the new value for pin 13. Essentially, the aim of this set-up is to replicate the behaviour of the signal generator using ASIP. The parameters that can be varied are:

- Type of connection: serial, TCP or MQTT.
- Software for the ASIP client: Java or Python language.
- Hardware where the client is running: Raspberry Pi or other machine. We have used a Macbook Pro and a Macbook Air (see below for details).
- In the case of networked connection: hardware and software configuration of the bridge.
- In the case of MQTT: broker location. Notice that we employ MQTT QoS level 0 ("at most once").

We report detailed experimental results in the following section.

4.1.2. Results

We present throughput experimental results separately for Java and for Python clients. The possible hardware configurations are:

- MacBook Pro: 2.4 GHz Intel Core i7, 16 GB of RAM, running Mac OS X 10.10.
- MacBook Air: 1.7 GHz Intel Core i5, 4 GB of RAM, running Mac OS X 10.8.
- RPi 2: Raspberry Pi, 900MHz quad-core ARM CPU, 1GB RAM, running the default Raspbian Linux image.
- Micro-controller: Arduino Uno running the default ASIP code available at:

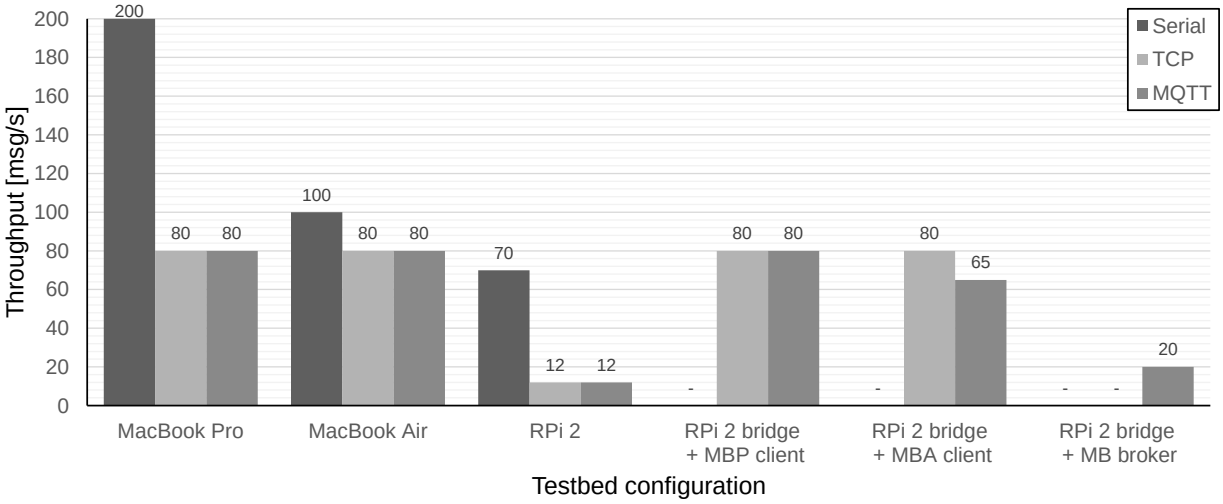
  `https://github.com/michaelmargolis/asip`

Additionally, in terms of network architecture for the experiments, we employ the following abbreviations:

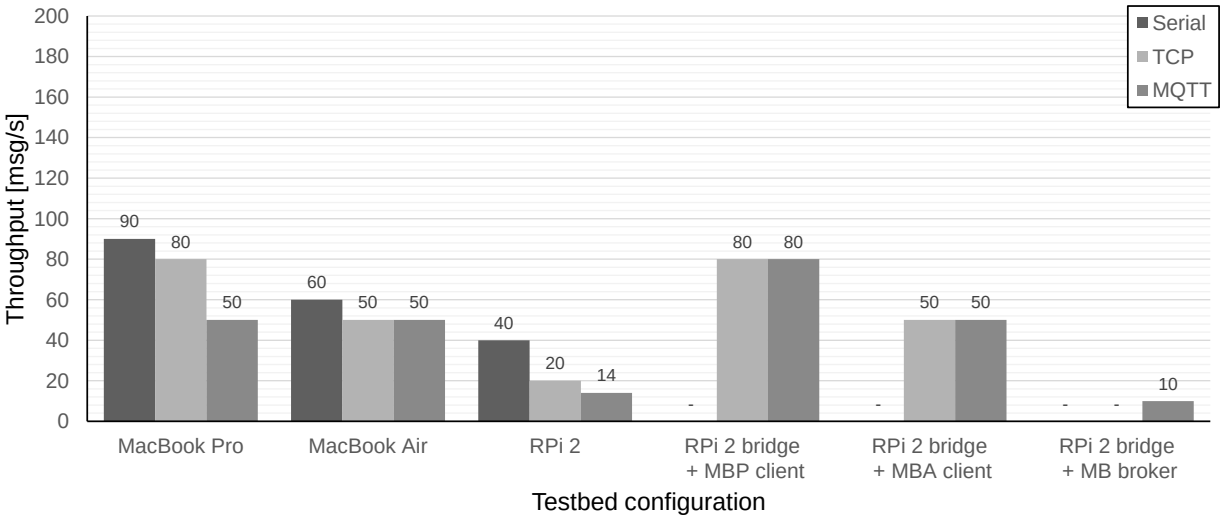
- MacBook Pro, MacBook Pro Air: the client and the TCP bridge (or MQTT broker) all run on the same machine to which the Arduino is connected using a USB cable.
- RPi 2: the client runs on a Raspberry Pi 2. In the case of TCP or MQTT connections, the bridge or the broker run on a *separate* Raspberry Pi. The two Raspberry Pi are connected using ethernet cables and a router.
- RPi 2 bridge + MBP client: in this configuration the software client runs on the MacBook Pro while the TCP bridge (or the MQTT broker) runs on the Raspberry Pi. Connection is through a router and ethernet cables.
- RPi 2 bridge + MBA client: as above, but the client runs on the MacBook Air.
- RPi 2 + MB broker: the MQTT broker runs on the MacBook Air and the client runs on the Raspberry Pi.

Figure 6a presents the experimental for the assessment of throughput using Java clients. In this set-up we employ the code available at `https://github.com/fraimondi/java-asip`. The serial library is provided by JSSC (`https://code.google.com/p/java-simple-serial-connector/`) while the MQTT library is provided by the Paho Java client (`https://eclipse.org/paho/clients/java/`). The MacBook Pro runs Oracle JVM 1.8, the MacBook Air and Raspberry Pi run Oracle JVM 1.6.

Figure 6b presents the experimental for the assessment of throughput using Python clients. In this set-up



(a)



(b)

Figure 6: Throughput for Java (a) and Python (b) clients with various testbed network configurations.

we employ the code available at `https://github.com/gbarbon/python-asip`. The serial library is provided by pySerial (`http://pyserial.sourceforge.net/`) while the MQTT library is provided by the Paho Python client (`https://eclipse.org/paho/clients/python/`). All experiments have been run using Python 2.7 but notice that the code is compatible with Python 3.

Discussion: The results presented in figures 6a and 6b show that ASIP can achieve a rate of messages up to 200 messages per second when the serial connection is used. In this case the limiting factor is the CPU speed of the client. As expected, TCP and MQTT performance is inferior to direct serial communication, but it is still more than adequate even for applications that require continuous monitoring, such as controlling a robot. The reduction in throughput is associated to the multiple communication layers introduced by the network libraries. Java outperforms Python in all tests; we argue that this is caused by the better performance of the Oracle JVM and its Just-in-Time compiler with respect to the Python interpreter. Interestingly, the throughput for TCP and MQTT connections is similar, with only minor differences in some circumstances. As mentioned above, MQTT connections are run at QoS level 0, and therefore there is not guarantee of message delivery, while TCP connections have built-in retransmission and sequencing guarantees. On the other hand, MQTT messaging allows broadcasting and the easy deployment of sensor networks.

Overall, we consider these results extremely promising and, in the case of serial connections, very close to the physical capacity of the communication channel, as explained in the following sections.

4.2. Latency

Testing for latency is performed using a single board. At a very high level, the test is performed by connecting an output pin with an input pin on the board, and then writing an ASIP client that sets the first pin to high and waits for a notification for the change of value of the second pin. The time difference between setting the output pin and measuring the change in the input pin is assumed to be the latency.

Similarly to the throughput test, we perform an assessment for serial connections (see Figure 7a) and for networked architectures using either TCP or MQTT (see Figure 7b).

For each one of the configurations described above we perform 100 tests and we take the average value.

4.2.1. Latency results

As in the case of throughput, we perform latency measures both for Java and for Python clients. The results for Java are reported in Figure 8a, while the results for Python are reported in Figure 8b.

In nearly all case, with the exception of two configurations running on resource-limited Raspberry Pi, the latency remains below 15 ms. We consider these very positive results, as the

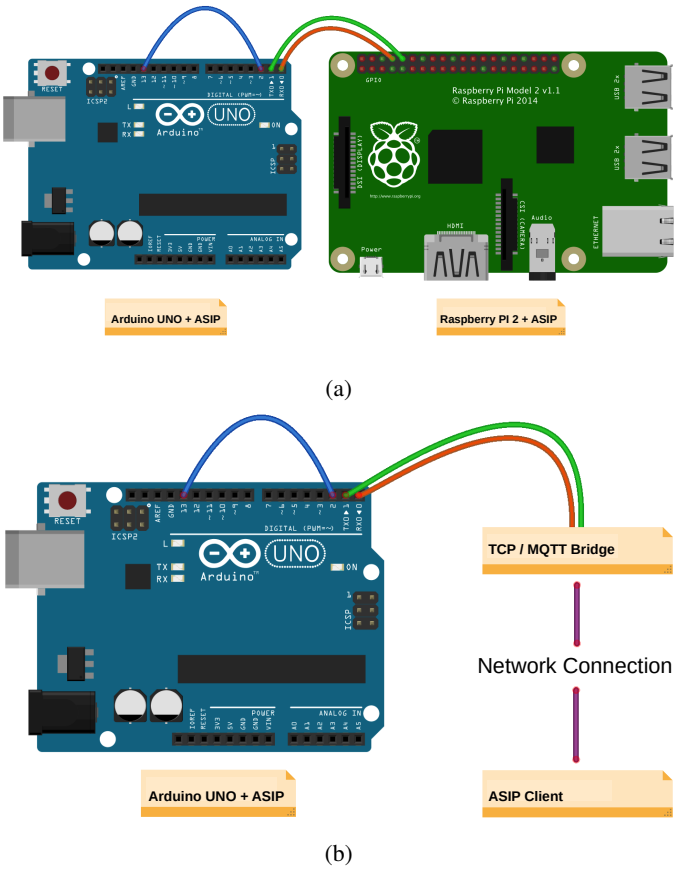


Figure 7: The hardware set-ups used for latency testing: (a) serial connection and (b) TCP and MQTT connection.

physical limitations of the serial communication channel introduce a latency of approximately 6.7 ms. This figure is computed by considering that 32 ASCII characters are exchanged in the ASIP messages for this application, by considering the additional bits required in each serial frame, and by considering the serial speed of 57600 baud.

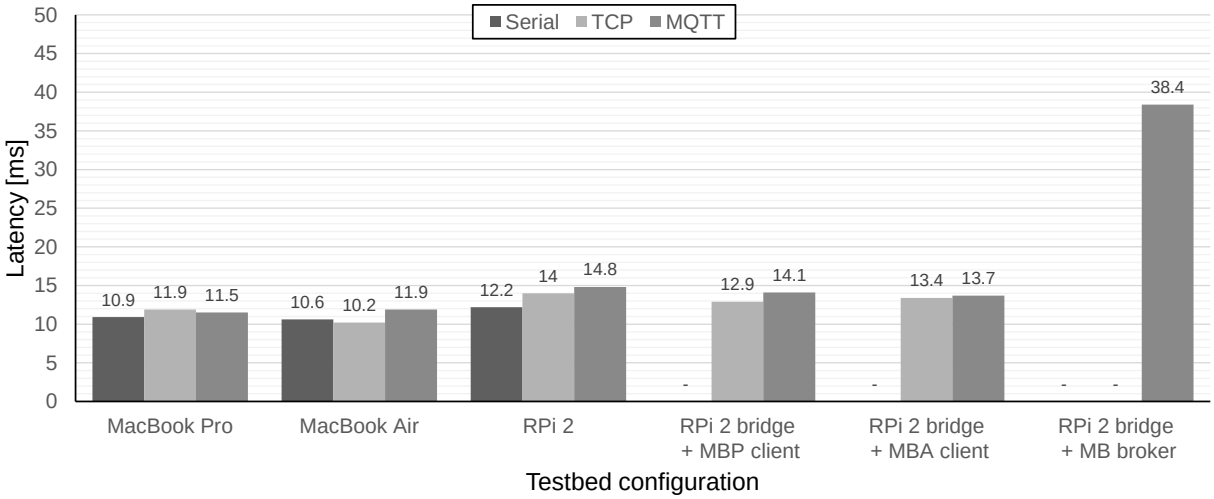
It is interesting to notice that latency is only minimally affected by the choice of the programming language and by the communication channel.

5. Qualitative Evaluation

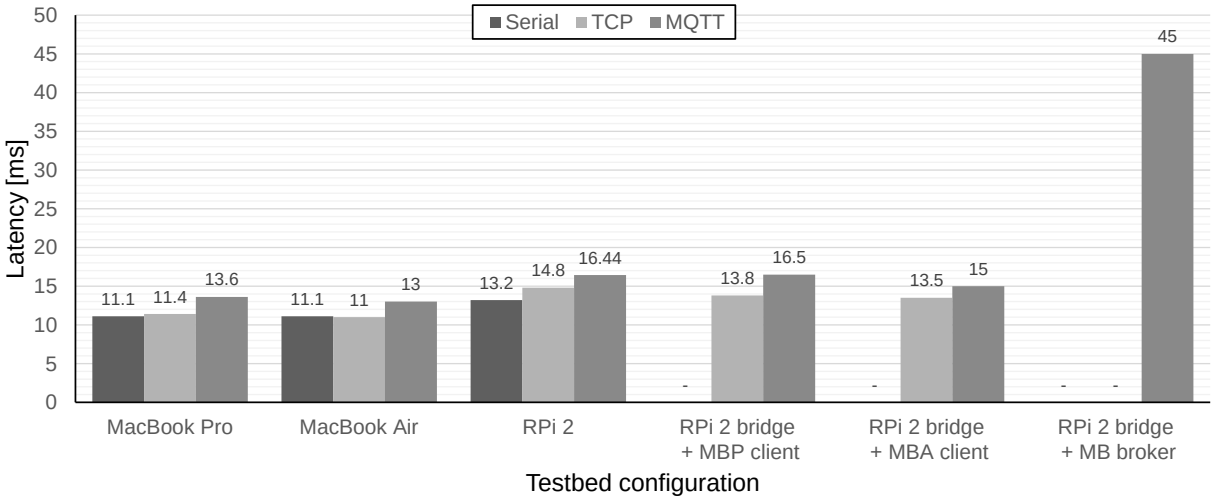
In this section we provide examples of how ASIP can be used and extended: we present how a simple distributed application can be built in Python, how a robot can be driven using a Proportional-Integral-Derivative (PID) controller [53] using Java and, finally, we show how to add a new service, both on the micro-controller and on the client code.

5.1. Building a distributed application

In this section we build a simple distributed application to coordinate two Arduino boards connected to the network using MQTT. In particular, an input button is connected to a board, and a LED is connected to the other board. Each board employs



(a)



(b)

Figure 8: Latency for Java (a) and Python (b) clients with various testbed network configurations.

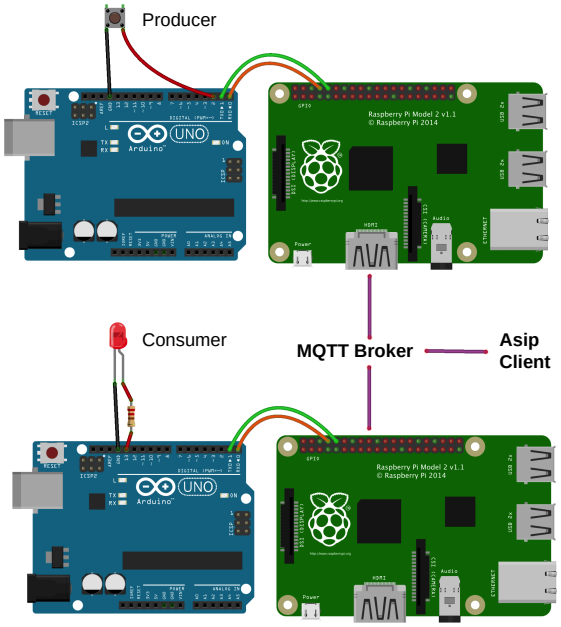


Figure 9: Button and LED application

a Raspberry Pi as a bridge to connect to an MQTT broker, as depicted in Figure 9. The aim of the application is to turn on the LED on the second board when the input button is pressed on the first board. The logic of the application is implemented by an ASIP client (not depicted in the figure). We assume that the Arduino boards have the standard I/O service installed and that there is an MQTT broker in the network with IP address 192.168.0.1.

Listing 1: Python example code for two boards connected using MQTT

```
1 from asip_client import AsipClient
2 from simple_mqtt_board import SimpleMQTTBoard
3  [...]
4
5 # A simple board with just the I/O services on a fixed port.
6 #  The main method simulates a light switch.
7 class TwoBoardSwitch():
8
9   def __init__ (self, Broker):
10     self.board1 = SimpleMQTTBoard(Broker, "board1")
11     self.board2 = SimpleMQTTBoard(Broker, "board2")
12
13     self.buttonPin = 2 # the pin on board1 for the input button
14     self.ledPin = 13  # the pin on board2 for the output (LED)
15     #  initialise  the variable  for  when we press the button
16     self.buttonState = 0
17     self.oldstate  = 0
18
19   def init_conn(self):
20     try:
21       [...]
22       self.board1.set_pin_mode(self.buttonPin, AsipClient.INPUT_PULLUP)
23       self.board2.set_pin_mode(self.ledPin, AsipClient.OUTPUT)
24       [...]
25     except Exception as e:
26       sys.stdout.write("Exception: caught {} in setting pin modes\n".format(e))
27
28   def main(self):
29     while True:
30       # check the value of the pin
31       self.buttonState = self.board1.digital_read(self.buttonPin)
32
33       # check if the value is changed with respect to previous iteration
34       if self.buttonState != self.oldstate:
35         if self.buttonState ==1:
36           self.board2.digital_write(self.ledPin, 1)
37         else:
38           self.board2.digital_write(self.ledPin, 0)
39         self.oldstate = self.buttonState
40
41 if __name__ == "__main__":
42   Broker = "192.168.0.1"
43   TwoBoardSwitch(Broker)
```

From a practical point of view, the developer needs to install ASIP on the two Arduino boards from `https://github.com/michaelmargolis/asip`, set up the two Raspberry Pi using either the Python or the Java MQTT bridges and, finally, implement an ASIP client. This client can run on a machine or on one of the two Raspberry Pi depicted in the figure. Excerpts for a Python client are presented in Listing 1. This client defines a class (line 7) composed of two boards of class `SimpleMQTTBoard` (lines 10 and 11), whose implementation is provided by the Python library available on github. Each board needs to be connected to a Broker (specified on line 42) and is identified by an ID. After initialising environment variables (lines 13 to 17) and setting the pins to appropriate modes (lines 22 and 23), the logic of the application is implemented in the loop of the main class between lines 29 and 39. The loop simply reads the state of the input pin on board1 (line 31). If the state has changed, then the state of the LED is changed appropriately (lines 36 or 38).

Notice how the communication mechanism is abstracted in this implementation. The physical location and the connection mechanism is irrelevant for the logic of the application in the main method: the only modification required to support a serial connection (or a TCP connection) is the definition of the boards on lines 10 and 11.

The expected performance in terms of latency and throughput for this kind of applications has been discussed in Section 4.

5.2. Controlling a Robot

In this section we show how a robot can be controlled over a TCP connection. In particular, we present the code to implement a line following algorithm for the Middlesex Robotic Platform (MIRTO) [54]. For the purposes of this example, we employ three infrared sensors mounted under the robot and we exploit the ability of controlling each wheel individually to the desired speed. The line to follow is a strip of black electric tape on a white table. When the infrared sensors are on the white surface they report a value close to 0, while when the sensors are perfectly on black they report values close to 1000. Any value in the 0-1000 range represent a partial overlap of the sensor with the black tape.

The robot is equipped with a bespoke PCB using an Atmel 328P chip, compatible with an Arduino Mini. The chip runs a version of ASIP including, in addition to the standard I/O service, services for infrared sensors and for wheel control. The Arduino code for this robot is available at `https://github.com/michaelmargolis/asip` in the sketch called `mirto.ino`. The ASIP client for this service are available at `https://github.com/fraimondi/java-asip` in the class JMirtoRobot.

The idea of a proportional-integral-derivative controller (PID) is that an error can be computed from the reading of the

infrared sensors. The greater the error, the farther away the robot from the line. We set a target speed and the correction of this speed, for each wheel, is proportional to three components:

- The *current* error (Proportional component)
- The *rate of change* of the error (the Derivative component)
- The *sum of the errors* so far (the Integral component).

This is normally capture by the following formula:

$$C(t) = K_p e(t) + K_i \int_0^t e(\tau)d\tau + K_d \frac{d}{dt} e(t)$$

where $C(t)$ is the correction at time t, $e(t)$ is the error computed at time t and K_p, K_i, K_d are the coefficients for the three components of the correction mentioned above.

Excerpts from the Java implementation for the PID algorithm are presented in Listing 2. The key points here are:

- Line 6 sets up a TCP connection to a bridge at IP address 192.168.0.1. This is the IP address of the Raspberry Pi running on the robot.
- The values of the infrared sensors are read at lines 10-12. The method `getIR` is provided by the class `JMirtoRobotOverTCP` that is available at `https://github.com/fraimondi/java-asip`.
- The values of the infrared sensors are used at line 16 and 20-21 to compute, respectively, the current error and the correction to be applied to the motors.
- The speed of each wheel is updated at line 23 with the method `setMotors`, which is provided by `JMirtoRobotOverTCP`

Listing 2: Java ASIP implementation for a PID line following robot

```
1 import uk.ac.mdx.cs.asip.JMirtoRobotOverTCP;
2
3 public class AsipMirtoPIDFollower {
4
5   public void navigate() {
6     JMirtoRobotOverTCP robot = new JMirtoRobotOverTCP("192.168.0.1");
7     // Additional code to set up pins goes here
8     // [...]
9     while (true) {
10      int leftIR = robot.getIR(2);
11      int middleIR = robot.getIR(1);
12      int rightIR = robot.getIR(0);
13
14      // computeError is a method to compute the current error as
15      // described above.
16      curError = computeError(leftIR,middleIR,rightIR);
17
18      // computeCorrectionLeft and Right computes the correct speed for
19      // each motor using the coefficient described above
20      speedLeft = computeCorrectionLeft(curError,Kp,Ki,Kd);
21      speedRight = computeCorrectionRight(curError,Kp,Ki,Kd);
22
23      robot.setMotors(speedLeft,speedRight);
24    }
25  }
26 }
```

The Java code can be run on a client connected to the same network to which the robot is connected. A typical line following algorithm performs smoothly at a frequency of approximately 10 Hz, which can be easily achieved given the results presented in the previous section for throughput. A video of this example is available at this link: `https://www.youtube.com/watch?v=KH_3766gNcM`

5.3. Adding new services

We conclude this section by showing how ASIP can be extended to include new services that are not yet available in our implementation. Adding a new service typically involves building code that runs on the Arduino and client code. In this section we show how the distance service can be implemented. The first step is to implement the class `AsipServiceClass` reported in Figure 1. A possible implementation is reported in Listing 3.

Listing 3: C++ code for the distance service

```
1 class asipDistanceClass : public asipServiceClass
2 {
3 public:
4    asipDistanceClass(const char svcId);
5    void begin(byte nbrElements, const pinArray_t pins[]);
6    // send the value of the given device
7    void reportValues(int sequenceId, Stream * stream);
8    void processRequestMsg(Stream *stream);
9 private:
10   int getDistance(int sequenceId);
11 };
12
13 // [...]
14 void asipDistanceClass::processRequestMsg(Stream *stream)
15 {
16   int request = stream->read();
17   [...]
18   if (request == 'M') {
19     reportValues(stream); // send a single measurement
20   }
21   [...]
22 }
23
24 void asipDistanceClass::reportValues(int sequenceId, Stream * stream)
25 {
26   if ( sequenceId < nbrElements) {
27     stream->print(getDistance(sequenceId));
28   }
29 }
30
31 int asipDistanceClass::getDistance(int sequenceId)
32 {
33   [...]
34   // The sensor is triggered by a HIGH pulse of 2 or more microseconds.
35   // Give a short LOW pulse beforehand to ensure a clean HIGH pulse:
36   byte pin = pins[sequenceId];
37   pinMode(pin, OUTPUT);
38   digitalWrite (pin, LOW);
39   delayMicroseconds(4);
40   digitalWrite (pin, HIGH);
41   delayMicroseconds(10);
42   digitalWrite (pin, LOW);
43   pinMode(pin, INPUT);
44   long duration = pulseIn(pin, HIGH, MAX_DURATION);
45   int cm = (duration / 29) / 2;
46   return cm;
47 }
```

This code implements the class `asipDistanceClass`, extending `asipServiceClass`. As a result, the class inherits a number of methods required to communicate over a stream etc., and it only needs to implement the actual service to be delivered. Concretely, in Listing 3 this means defining a unique ID for the service (used in the constructor at line 4) and defining

the commands that can be sent to the service in the method `processRequestMsg`, as exemplified at line 18 where the distance service implements a response to the command 'M' by calling method `reportValues` which, in turn (line 27) calls the private method `getDistance`. This private method (lines 31 to 47) is the method that implements the actual service and in the case of other services it may employ specific libraries for a service, such as for controlling wheels. By compiling this code and uploading it to the micro-controller it is now possible to send ASIP messages of the form `D,M`: these will be captured by the `processRequestMsg` method that, in turn, will generate a response of the form `@D,e,1,{0:42}` meaning that this is a message from a Distance service reporting a distance event; there is only one distance sensor attached and that its current reading is 42 cm. Additional sensors can be added without any modification of the code, see line 26 in Listing 3.

Typically, clients are extended to support the new services deployed on the micro-controller. As an example, Listing 4 shows excerpts from the Java client for the distance service described above. The full code is available in the file `DistanceService.java` at `https://github.com/fraimondi/java-asip`.

Listing 4: Java client for the distance service (excerpts)

```
1 public class DistanceService implements AsipService {
2
3   private char serviceID = 'D';
4   // This is the last measured distance (-1 if not initialised )
5   private int lastDistance;
6   //  [...]
7
8   public void requestDistance() {
9     this.asip.getAsipWriter().write(this.serviceID+","+'M'+"\n");
10  }
11  // [...]
12
13  public void processResponse(String message) {
14    if (message.charAt(3) == DISTANCE_EVENT) {
15      String distances = message.substring(message.indexOf("{")+1,
16          message.indexOf("}"));
17      this.lastDistance = Integer.parseInt(distances.split(",")[this.distanceID]);
18    }
19  }
20 }
```

As in the case of the code running on the micro-controller, the Java class extends a superclass that provides most of the methods already. The new subclass only needs to implement the method to process responses, see line 15. This new class is then ready to be used in conjunction with ASIP applications. Notice how the developer only needs to implement the specific feature of a service and can re-use all the networking infrastructure.

6. Conclusion

In this paper we have introduced the Arduino Service Interface Programming model (ASIP). This is an infrastructure that comprises:

- a software architecture to manage micro-controllers as clients of higher-level languages;
- a language for messages exchanged over a range of communication channels between micro-controllers and clients;
- a communication and network architecture that can be based on direct serial (USB) links, TCP sockets, and MQTT publish/subscribe messaging.

We have provided a concrete implementation for Arduino micro-controllers and libraries for a range of programming languages. All our source code has been released as open source. We have performed an extensive assessment of the performance of the ASIP infrastructure using Java and Python clients both direct serial connections and over networked connections. The results obtained are very encouraging and show that latency and throughput are adequate for controlling precise navigation of a robot over a wireless network. Besides performance consideration, we have provided a qualitative evaluation showing how applications can be developed by exploiting the existing libraries using only a few lines of code and delegating the communication and coordination issues among microcontrollers to the underlying ASIP infrastructure.

For the future, we are currently working on the implementation of ASIP bridges based on the ESP8266 chip. While this paper has focussed on the practical implementation for Arduino micro-controllers, we remark that the service model described is independent from the actual micro-controller hardware. The only requirement is that the micro-controller should support a communication stream and support execution loops.

From a system and software engineering point of view we consider this work a first step in the direction of model-based development for complex applications involving multiple micro-controllers. Our aim is to enable automatic code generation from our service model, working in the direction of verification and certification activities for complex domains [55].

References

[1] L. Atzori, A. Iera, G. Morabito, The internet of things: A survey, Computer networks 54 (15) (2010) 2787–2805.

[2] D. Guinard, V. Trifa, S. Karnouskos, P. Spiess, D. Savio, Interacting with the soa-based internet of things: Discovery, query, selection, and on-demand provisioning of web services, Services Computing, IEEE Transactions on 3 (3) (2010) 223–235.

[3] D. Guinard, V. Trifa, F. Mattern, E. Wilde, From the internet of things to the web of things: Resource-oriented architecture and best practices, in: Architecting the Internet of Things, Springer, 2011, pp. 97–129.

[4] Z. Sheng, S. Yang, Y. Yu, A. Vasilakos, J. Mccann, K. Leung, A survey on the ietf protocol suite for the internet of things: Standards, challenges, and opportunities, Wireless Communications, IEEE 20 (6) (2013) 91–98.

[5] J. Kim, J. Lee, J. Kim, J. Yun, M2m service platforms: survey, issues, and enabling technologies, Communications Surveys & Tutorials, IEEE 16 (1) (2014) 61–76.

[6] Cosm, accessed: 2016-01-29.
URL `https://cosm.com`

[7] Thingspeak, accessed: 2016-01-29.
URL `https://www.thingspeak.com/`

[8] Nimbits, accessed: 2016-01-29.
URL `https://www.nimbits.com/`

[9] Evrythng, accessed: 2016-01-29.
URL `http://evrythng.com/`

[10] Oneplatform by exosite, accessed: 2016-01-29.
URL `http://exosite.com/`

[11] Axeda platform, accessed: 2016-01-29.
URL `http://www.axeda.com/`

[12] Sensorcloud, accessed: 2016-01-29.
URL `http://www.sensorcloud.com/`

[13] Neuaer, accessed: 2016-01-29.
URL http://www.neuaer.com/
[14] idigi device cloud, accessed: 2016-01-29.
URL http://www.digi.com/
[15] W. Colitti, K. Steenhaut, N. De Caro, B. Buta, V. Dobrota, Rest enabled wireless sensor networks for seamless integration with web applications, in: Mobile Adhoc and Sensor Systems (MASS), 2011 IEEE 8th International Conference on, IEEE, 2011, pp. 867–872.
[16] Z. Shelby, K. Hartke, C. Bormann, The constrained application protocol (coap).
[17] Sensinode, accessed: 2016-01-29.
URL http://www.sensinode.com/
[18] M. Collina, G. E. Corazza, A. Vanelli-Coralli, Introducing the qest broker: Scaling the iot by bridging mqtt and rest, in: Personal Indoor and Mobile Radio Communications (PIMRC), 2012 IEEE 23rd International Symposium on, IEEE, 2012, pp. 36–41.
[19] P. T. Eugster, P. A. Felber, R. Guerraoui, A.-M. Kermarrec, The many faces of publish/subscribe, ACM Computing Surveys (CSUR) 35 (2) (2003) 114–131.
[20] T. Sheltami, A. Al-Roubaiey, A. Mahmoud, E. Shakshuki, A publish/subscribe middleware cost in wireless sensor networks: A review and case study, in: Electrical and Computer Engineering (CCECE), 2015 IEEE 28th Canadian Conference on, IEEE, 2015, pp. 1356–1363.
[21] C. Intanagonwiwat, R. Govindan, D. Estrin, J. Heidemann, F. Silva, Directed diffusion for wireless sensor networking, IEEE/ACM Transactions on Networking (ToN) 11 (1) (2003) 2–16.
[22] E. Souto, G. Guimarães, G. Vasconcelos, M. Vieira, N. Rosa, C. Ferraz, J. Kelner, Mires: a publish/subscribe middleware for sensor networks, Personal and Ubiquitous Computing 10 (1) (2006) 37–44.
[23] P. Levis, S. Madden, J. Polastre, R. Szewczyk, K. Whitehouse, A. Woo, D. Gay, J. Hill, M. Welsh, E. Brewer, et al., Tinyos: An operating system for sensor networks, in: Ambient intelligence, Springer, 2005, pp. 115–148.
[24] J.-H. Hauer, V. Handziski, A. Köpke, A. Willig, A. Wolisz, A component framework for content-based publish/subscribe in sensor networks, in: Wireless Sensor Networks, Springer, 2008, pp. 369–385.
[25] O. OMG, Data distribution service for real-time systems, Tech. rep., Technical Report OMG Available Specification formal/07-01-01, OMG (2006).
[26] P. Boonma, J. Suzuki, Self-configurable publish/subscribe middleware for wireless sensor networks, in: Proceedings of the th IEEE Conference on Consumer Communications and Networking Conference (CCNC), pp.–, IEEE Press, 2009.
[27] A. Castellani, N. Bui, P. Casari, M. Rossi, Z. Shelby, M. Zorzi, Architecture and protocols for the internet of things: A case study, in: Pervasive Computing and Communications Workshops (PERCOM Workshops), 2010 8th IEEE International Conference on, IEEE, 2010, pp. 678–683.
[28] Q. Zhu, R. Wang, Q. Chen, Y. Liu, W. Qin, Iot gateway: Bridgingwireless sensor networks into internet of things, in: Embedded and Ubiquitous Computing (EUC), 2010 IEEE/IFIP 8th International Conference on, IEEE, 2010, pp. 347–352.
[29] U. Hunkeler, H. L. Truong, A. Stanford-Clark, Mqtt-sa publish/subscribe protocol for wireless sensor networks, in: Communication systems software and middleware and workshops, 2008. comsware 2008. 3rd international conference on, IEEE, 2008, pp. 791–798.
[30] M. Franklin, S. Zdonik, data in your face: push technology in perspective, in: ACM SIGMOD Record, Vol. 27, ACM, 1998, pp. 516–519.
[31] A. Stanford-Clark, H. L. Truong, Mqtt for sensor networks (mqtt-sn) protocol specification (2013).
[32] iobridge - connect things, accessed: 2016-01-29.
URL http://iobridge.com/
[33] Lord sensing microstrain, accessed: 2016-01-29.
URL http://www.microstrain.com/
[34] Arduino, accessed: 2016-01-29.
URL http://www.arduino.cc/
[35] Armmbed, accessed: 2016-01-29.
URL http://mbed.org/
[36] nanode, accessed: 2016-01-29.
URL http://www.nanode.eu/
[37] Lilypad arduino main board, accessed: 2016-01-29.
URL https://www.arduino.cc/en/Main/ArduinoBoardLilyPad/
[38] H. G. Cerqueira Ferreira, E. Dias Canedo, R. T. De Sousa, Iot architecture to enable intercommunication through rest api and upnp using ip, zigbee and arduino, in: Wireless and Mobile Computing, Networking and Communications (WiMob), 2013 IEEE 9th International Conference on, IEEE, 2013, pp. 53–60.
[39] P. Barsocchi, E. Ferro, F. Palumbo, F. Potorti, Smart meter led probe for real-time appliance load monitoring, in: SENSORS, 2014 IEEE, IEEE, 2014, pp. 1451–1454.
[40] A. Al-Fuqaha, M. Guizani, M. Mohammadi, M. Aledhari, M. Ayyash, Internet of things: A survey on enabling technologies, protocols, and applications, Communications Surveys & Tutorials, IEEE 17 (4) (2015) 2347–2376.
[41] D. Thangavel, X. Ma, A. Valera, H.-X. Tan, C. K.-Y. Tan, Performance evaluation of mqtt and coap via a common middleware, in: Intelligent Sensors, Sensor Networks and Information Processing (ISSNIP), 2014 IEEE Ninth International Conference on, IEEE, 2014, pp. 1–6.
[42] W. Colitti, K. Steenhaut, N. De Caro, B. Buta, V. Dobrota, Evaluation of constrained application protocol for wireless sensor networks, in: Local & Metropolitan Area Networks (LANMAN), 2011 18th IEEE Workshop on, IEEE, 2011, pp. 1–6.
[43] S. Nicholas, Power profiling: Https long polling vs. mqtt with ssl, on android, accessed: 2016-01-29.
URL http://stephendnicholas.com/archives/1217/
[44] H. W. Chen, F. J. Lin, Converging mqtt resources in etsi standards based m2m platform, in: Internet of Things (iThings), 2014 IEEE International Conference on, and Green Computing and Communications (GreenCom), IEEE and Cyber, Physical and Social Computing (CPSCom), IEEE, IEEE, 2014, pp. 292–295.
[45] P. Derler, E. Lee, A. S. Vincentelli, et al., Modeling cyber–physical systems, Proceedings of the IEEE 100 (1) (2012) 13–28.
[46] M. Quigley, K. Conley, B. Gerkey, J. Faust, T. Foote, J. Leibs, R. Wheeler, A. Y. Ng, Ros: an open-source robot operating system, in: ICRA workshop on open source software, Vol. 3, 2009, p. 5.
[47] H.-C. Steiner, Firmata: Towards making microcontrollers act like extensions of the computer, in: New Interfaces for Musical Expression, 2009, pp. 125–130.
[48] Particle (formerly spark) — build your internet of things, accessed: 2016-01-29.
URL https://www.particle.io/
[49] F. Palumbo, J. Ullberg, A. Štimec, F. Furfari, L. Karlsson, S. Coradeschi, Sensor network infrastructure for a home care monitoring system, Sensors 14 (3) (2014) 3833–3860.
[50] F. Palumbo, P. Barsocchi, F. Furfari, E. Ferro, Aal middleware infrastructure for green bed activity monitoring, Journal of Sensors 2013.
[51] F. Palumbo, D. La Rosa, S. Chessa, Gp-m: Mobile middleware infrastructure for ambient assisted living, in: Computers and Communication (ISCC), 2014 IEEE Symposium on, IEEE, 2014, pp. 1–6.
[52] P. Barsocchi, E. Ferro, L. Fortunati, F. Mavilia, F. Palumbo, Ems@ cnr: An energy monitoring sensor network infrastructure for in-building location-based services, in: High Performance Computing & Simulation (HPCS), 2014 International Conference on, IEEE, 2014, pp. 857–862.
[53] K. H. Ang, G. Chong, Y. Li, Pid control system analysis, design, and technology, Control Systems Technology, IEEE Transactions on 13 (4) (2005) 559–576. doi:10.1109/TCST.2005.847331.
[54] K. Androutsopoulos, N. Gorogiannis, M. J. Loomes, M. Margolis, G. Primiero, F. Raimondi, P. Varsani, N. Weldin, A. Zivanovic, A Racket-based robot to teach first-year computer science, in: 7th European Lisp Symposium, 2014, pp. 54–61.
URL http://eprints.mdx.ac.uk/14027/
[55] RTCA, DO-178C, software considerations in airborne systems and equipment certification (2011).