Low-Cost Robots in the Internet of Things: Hardware, Software & Communication Aspects

L. Dauphin
INRIA Paris-Saclay

H. Petersen
Freie Universität Berlin

C. Adjih and E. Baccelli
INRIA Paris-Saclay

Abstract

In the future, the Internet will not only connect computers and smart objects, but also a wide variety of semi- and fully-autonomous robots. This paper focuses on low-cost robots, and studies the recent convergence between low-cost robot hardware and IoT hardware. The potential for further convergence is then explored in terms of common embedded software platforms and common communication protocols, which could be used on both low-end IoT devices and low-cost robots. Finally, a proof-of-concept is provided based on RIOT and Aversive++ as software platform, show-cased running on a low-cost four-legged IoT robot.

1 Introduction

Robots are increasingly integrated in our daily lives, starting with cleaning robots, and self-driving cars. In the near-future, the density of robots is likely to increase dramatically. Swarms of fully or semi-autonomous low-cost robots will interact locally with humans and machines in the vicinity, as well as remotely, through available communication networks. In this context, the Internet of Things (IoT) and robotics are likely to merge into a continuum whereby no fundamental difference is made between low-cost robots and collaborative smart objects. In this paper, we will denominate this continuum as *IoT robotics*, and we explore the potential use of low-end IoT devices hardware, software and communication to realize IoT robotics.

1.1 Low-cost robots

In the scope of IoT robotics, traditional types of robots can be considered as connected object, and even as a collection of connected objects assembled together to achieve several missions.

Wheeled robots are the simplest, composed generally from 2 to 4 motored wheels. Each wheel can be controlled in speed, and the wheels of the robots can be controlled together to create a smart movement. In the first case, each controlled wheel can be considered as a smart object that can be assembled to a non-mobile object to transform it into a connected robotic base. In the second case, the base itself is already a connected object, which can be assembled to a payload to complete it's mission. For example, it can embed a robotic arm to be able to interact more precisely with its environment. Or it can embed a connected sensor to monitor an area. Wheeled robots range from self-driving vehicles to extremely cheap mini-robots, approximately of the size and computing capabilities of current IoT devices (see for instance [35], and [11]), which are expected to become commodity and 1000 times cheaper than currently available robots, for instance the AFRON Challenge [28] has shown that a 10$ price tag is possible for tiny wheeled robots.

Robotic arms are more complex and more expensive. Composed of several servomotors, they need complex algorithms to be controlled. Modern servomotors for robotics tend to be smarter, they can be connected to a bus, and use a communication protocol, which makes them kind-of connected objects. Robotic arms are not mobile robots, so that either their mission is limited to the surrounding area, or it must be embedded on a mobile robot. Leveraging a number of emerging techniques, such as 3D printing (see for instance as Poppy, an open source, 3d printed, robot [31, 29]), very cheap robotic arms are already available.

Legged robots increase complexity and price, compared to robotic arms. Composed of a set of robotic arms (considered as legs, in this case), they also have a strong constrain on timing to walk properly. Most legged robots are either toys or research platforms, because their movement is not yet mastered enough to be deployed autonomously in end-user homes. The number of required servomotors makes them quite expensive for their current use, but as for robotic arms, their price tend to be lower and lower [9].

Other low-cost robots of course include an ever-increasing variety of drones, but may also include entirely new types of robots in the future, which could as well benefit from the merge of IoT and robotics. One particular example is self-configuring robots [5, 36]. More generally: any set of sensors and actuators collaborating locally and/or with remote controllers (e.g. some logic in the cloud) may to some extent be considered equivalent to a robot. In this context, it follows that low-cost robots and other IoT devices could share a common base of hardware modules, software platforms and communication protocols.

This paper explores this space, focusing primarily on low-cost (ground) robots. The contributions of this paper are the following. First, we show the convergence between low-cost robot hardware and IoT hardware. Second, we explore the potential for convergence of embedded software platforms and communication protocols used on low-end IoT devices and on low-cost robots. Last, we describe first steps and provide a proof-of-concept for this convergence, based on RIOT and Aversive++ as software platform running on a low-cost legged robot.

International Conference on Embedded Wireless Systems and Networks (EWSN) 2017
20–22 February, Uppsala, Sweden

ISBN: 978-0-9949886-1-4

1.2 Related work

Cloud robotics [27] envisions massive deployments of robots with limited computing power, enhanced by communicating with remote (cloud) computing capacities. In [33], authors demonstrate a cheap IoT-controlled car with some behavior logic exported in the cloud. Swarms of low-cost robots were studied previously in the literature. For example, in [22]) a practical approach to low-cost swarm robotics is studied, while [20] surveys theoretical and algorithmic frameworks for swarm robotics. On the other hand [25] surveys IoT-aided robotics.

2 Low-Cost Robot Hardware

This section quickly overview the main hardware parts of a low-cost ground robot, and highlights convergence between low-cost robotics hardware and low-end IoT hardware.

2.1 Actuators

Robots embark two types of actuators: (i) actuators that are part of the robot itself, and (ii) payload actuators, that are carried by the robot to fulfill a mission. There is no difference between these two types, but the use that is made of the actuator.

Actuators for low cost robotics are, essentially : continuous rotation motors and angle controlled motors. Associated with mechanical parts, these two actuators can achieve almost any movement.

Continuous rotation motors are typically used to make wheeled mobiles robots. The most common are continuous current brushed motors, which are inexpensive and simple to control. Brushless motors (i.e. BLDC/PMSM) can be much more efficient both in terms of effectiveness and in their power-to-weight ratio, but need sophisticated electronics and algorithms to be controlled. Angle controlled stepper motors typically have rather large holding torque but are limited to their revolution speed, and can be interfaces without much complexity. Depending on the technology and quality of the motor, they are available costing from 10s of cents to 100s of euros.

The angle-controlled motors are generally simple motors as described above, but with some additional sensors and electronics to be controlled in angle. Angle-controlled motors are typically used for robotic arms. They can be sorted in 2 categories, depending on the mode of communication of the control system: smart servomotors and simple servomotors. Simple servomotors can only receive PWM (Pulse-Width Modulated) signals, which represents the angle command. Smart servomotors communicate via more complex protocols, such as a Half-duplex UART bus. Smart servomotors can receive angle commands, but can also send the actual angle of the actuators, enable the configuration of the control loop, and other features. While most simple servomotors typically cost from 5 to 40 euros, smart servomotors typically range from 30 to 300 euros. Dynamixel smart servomotors are the most widely used in robotics.

The energy consumption of servomotors varies significantly depending on the actuator's quality, the maximum speed and maximum torque. For example, simple (standard sized) servomotors typically consumes 250mA on average and 2A if used at their maximum torque.

The price and number of actuators of a robot (in particular, the price of its smart servomotors) generally determines a huge chunk of the robot's price, typically more than 50%.

2.2 Sensors

Robots embark two types of sensors: (i) sensors that are part of the robot itself, and (ii) payload sensors, that are carried by the robot to fulfill a mission.

Sensors for the robot are sensors that helps the robot to control it's movement and know it's position in it's environment. For example, the servomotors contains generally angle sensors that let the robot know its body's pose. The angle sensors can be simple potentiometers, but other technologies can be used. Another example is a rotary encoder used by wheeled robots to measure the distance traveled by the wheels. Some robots use IMUs (Inertial Measurement Unit) to measure the distance traveled by it's body and it's heading. Other (outdoors) robots also use GPS to know their position. Robots can also use a wide variety of sensors to avoid obstacles in their environment: bumpers, light/sound powered distance sensors, cameras, radar, lidar...

Payload sensors include a broad category of sensors, depending of the mission of the robot : cameras for survey or area mapping, weather sensors (temperature, pression, light), presence sensors, gas sensors, etc... Note that some sensors can be common between payload and sensors for the robot.

2.3 Power Supply

Compared to low-power nodes in the IoT, robots draw a significantly larger amount of electrical power. This is mainly due to their actuators (i.e. motors) and duty cycles, IoT devices sleep most of their time, while robots need to be active at least while they are moving. Since the ratio of energy usage caused by the actuators is very large compared to standard micro-controllers and most sensors, the power consumptions of the two latter is in many cases negligible.

In contrary to most IoT nodes, weight plays a large role for many mobile robots, especially anything airborne. For this reason many different battery technologies (e.g. LiPo, LiIon, NiMh, NiCd, Lead) are being used. Distinguishing characteristics are next to the energy-to-weight ratio also e.g. maximum output current, temperature behavior, or reload cycles. While some alternative solutions like supercapacitors exists for some niches, there use is rather uncommon. Also energy harvesting, often used for very low-power wireless sensor applications, plays no significant role for the targeted low-cost robots.

2.4 Computing and Communication Unit(s)

Computing units embedded on a robot have generally two purpose (i) making the robot autonomous, and (ii) enabling some level of tele-operation. Depending of the robot's mission, one part can be more important than the other, or nonexistent. Autonomous robots will require more powerful hardware, to be able to run complex algorithms. Tele-operated robots will need a reliable communication with their operator. If an autonomous robot can't embark powerful hardware, combining communication and remote computing power can compensate lack of local computing power. However, in this case, the need for reliable communication is similar to that of tele-operated robots.

For example, many low-cost robots embark computing units such as the Raspberry Pi [31, 29], and use WiFi or Ethernet in terms of communication technology. On the other hand, low-cost robots also embark a variety of smaller units such as Arduino boards [5, 23], and use ZigBee, WiFi or Bluetooth. Such small units are preferred not only for their lower energy consumption (negligible when the robot is moving, but relevant when the robot is not moving) but rather but also to reduce the size and cost of the computing parts (especially for really small robots), and for improving the modularity of the robot. Indeed, a robot with several modules is more able to evolve than a robot with a high-end monolithic computing unit.

2.5 Convergence with IoT Hardware

In the IoT, two types of computing/communication units are typically used.

High-end units on one hand, are able to execute traditional operating systems, use traditional WiFi or Ethernet, with standards protocols of the Internet. They are characterized by MBytes of data memory (RAM) or more, GBytes of program memory (SD/Flash) and a clock frequency around 1GHz (or more). Examples of such units used in the IoT include the Raspberry Pi, Beaglebone, connected cameras, smartphones etc.

Low-end units, on the other hand, are usually classified in several categories, as specified in [19], based on their computing power and memory capacity:

- Class 0 devices embark around 1kBytes of data memory, a clock running at less than 16MHz, and less than 32kBytes of code memory. Well-known IoT devices in this class include for instance the Arduino UNO board. Such devices can barely run an operating system and are generally dedicated to the control of a single actuator or sensor. Class 0 devices cannot control complex systems, and rely on a more powerful computing unit with which it communicates with simple protocols enabling local communication, as Class 0 devices typically can't handle full blown network stacks.
- Class 1 devices embark around 10kBytes of data memory, a clock running at around 16MHz, and around 100kBytes of code memory. Well-known IoT devices of this class include for instance Arduino MEGA 2560 [1], BBC micro:bit [4], Atmel SAMR21 [2] or Zolertia ReMote [16], OpenMote [12] boards. Class 1 devices can run small operating systems such as RIOT [18] or other similar OS for low-end IoT devices [26], but not necessarily fully featured : one may be required to select features due to memory constraints. In particular, it is possible to use Internet protocols on such devices, at the price of a significant chunk of the capacity of the device.
- Class 2 devices embark around 50kBytes data memory, a clock running at 200MHz or less, and around 250kByte of code memory. Well-known IoT devices of this class include most devices based on ARM Cortex-M micro-controllers, for example the Eistec Mulle [6], IoT-lab M3 [8], Phytec phyNODE [13] or HiboB Fox [7] boards. Class 2 devices can run an OS for low-end IoT devices with all the needed features for IoT.

Low-end IoT devices use a variety of low-power communication technologies such as IEEE 802.15.4, BLE, LoRa, BACnet etc. on top of which a compressed version of the IPv6 protocol stack (the 6LoWPAN stack [30]) enables end-to-end communication with virtually any destination on the Internet.

IoT devices also embark a wide variety of sensors (temperature, humidity, light, noise...) and actuators (PWM driven light color actuators, HVAC, smart locks, light switch...) which communicate with the computation unit via bus protocols such as UART, I2C, SPI.

We can thus observe a striking convergence between low-cost robotics hardware described in previous sections. The following sections will outline how a similar convergence could happen concerning software platforms and communication protocols use on low-cost robotics and low-end IoT devices, focusing primarily on devices/unites in the class 1 and class 2 categories.

3 IoT Robotics Software

The software essential for running a robot can be loosely mapped to a four levels of functionality. Each level has different characteristics w.r.t. timing, processing power demands, and means to distribute the level over different computational units.

Level 1: The first and lowest level covers everything related to direct hardware access. This includes operating system (OS) services (e.g. tasks, mutexes), hardware abstraction (e.g. timers, UART, PWM), and device drivers for low-level interaction with the used sensors and actuators. On this level there are no means of control loops, so all functions are considered **non-periodically and very low latency**. Due to the direct interaction with the hardware, this level is fixed to a specific computing unit and can therefore **not be distributed**. The demands for processing power depend naturally on the number of sensors/actuators that are handled by this level, but are typically not very high and thus making this level feasible to run on **anything from Class 0 devices**.

Level 2: On the second level we locate the periodically running, low-level control algorithms. Typical are PID and similar controllers for motor control and sensor data handling algorithms, ranging from simple filters (e.g. average, differential, bang-bang) to more sophisticated Kalman filters for sensor fusion. Looking further at robotic arms (i.e. legs), also algorithms for solving the inverse kinematic (e.g. Jacobian method) are located on this level. The timing requirements on this level can be very rigid, where BLDC/PMSM motor controllers run with periods as low as 100s of nanoseconds, but typical are periods range in the magnitude from some **10s of microseconds to 100s of milliseconds**. While fast running control loops are not feasible to be distributed, it is possible or even necessary to distribute sensor data aggregation and fusion over multiple nodes, hence making this level **partly distributable**. The needed processing power on this level is bounded by the used period with which the algorithms are run as well as the complexity of the actual used algorithms. Class 0 devices are too limited for running things

like Kalman filter, Clark-Park transformations, or solving inverse kinematic problems, **demanding for Class 1 or better Class 2 devices** for things like multi-legged robots.

Level 3: This level takes care of trajectory planning and obstacle avoidance. This requires the robot to keep an internal representation of its environment. Depending on the complexity of the environment and the model, this can take up significant amounts of memory and processing power. Typical run on this level are SLAM (Simultaneous Localization And Mapping) algorithms based on Bayesian models as well as path finding algorithms like *A**. The refresh periods of these computations range typically in the magnitude from **10s of milliseconds to seconds**, making them generally *offloadable* to a more capable computing unit. A, potentially simplified and stripped down, instance of these algorithms needs to be carried out however locally at any time. This ensures that the robot can be put into a safe state in any point of time, saving it from collisions and similar in case its communication abilities fail. These simplified computations are **well suited to run on Class 1 devices**.

Level 4: The highest level of robot software houses the mission and task control. The control periods on this level are rather coarse, **ranging typically from minutes upwards to days and months**. This level **generally does not need to run on the robot entirely**, control software on this level either being distributed by definition (e.g. decentralized swarms), or being build around a centralized control unit. Focusing on the a centralized controlled setup, the needed computational resources are bounded by running the communication and by retrieving and buffering simple commands, and can hereby **easily be deployed on Class 1 devices**.

IoT software: Software running on constrained IoT devices shares the same characteristics with *level 1*. The demands on timing depends highly on the actual use-cases, and can range from **100s of milliseconds to minutes and hours** for e.g. light switches and or environmental data loggers, respectively. In contrary to robotics software, the actual application software is however **distributed by default**. While some applications can be implemented on Class 0 devices, **Class 1 devices are considered the minimum** when running full Internet connectivity including security.

3.1 Existing and Potential Platforms

As of today, from a software perspective most robots are build on top of middleware and libraries, that provide implementations many needed algorithms. Prominent examples are ROS [34], OROCOS [21], and OpenRAVE [24]. All of these cover the algorithms needed for *levels 2 to 4* (e.g. PID controllers, SLAM, *A**), while offering only limited support for device drivers. Thus they depend on an underlying operating system for most of *level 1* functionality, this typically being Linux or Windows. As these middlewares are not designed with a primary focus on constrained embedded systems, they can generally not be run on the platforms targeted by this paper.

At the same time, a number of real-time operating systems targeting specifically IoT applications emerged. These include RIOT [18], mbed OS [17], Zephyr [14] , Mynewt [10] and are explicitly designed for constrained devices that are not able to run full blown OSes as Linux. Furthermore these OSes were primarily designed for running distributed applications (e.g. Internet connectivity). They also provide more or less sophisticated hardware abstraction models, satisfying all the demands of *level 1* functionality.

Recently the Aversive++ library [3] was ported to RIOT, providing a unique new connection between an embedded real-time operating system designed for Internet connectivity with a robotics library providing large parts of the functionality needed for robots. This combination provides a platform being able to run on the targeted Class 1 and 2 satisfying the requirements defined above.

4 IoT Robotics Communication

As the success of the ROS framework shows, robotic applications benefit from being designed as a collection of loosely coupled modules (referred to as nodes in ROS), connected through some form of messaging system. This enables nodes to be deployed on different computing units, possibly even outside the robot. Thus, if a robot's computing power is exceeded, nodes can be deployed on external resources without being modified.

Distributing nodes among different computing unit connected by a network (e.g. WiFi, IEEE 802.15.4, Bluetooth etc.) opens however a new set of problems: messages are subject to data loss, jitter, extended delays, and so on. Considering the substantial limitations of low power wireless communications, IoT robotic algorithms need to be implemented in a way, so that they can cater with these newly introduced problems. In the following we overview communication requirements at various software levels (which were defined in section 3).

The communication between **levels 1 and 2** can be described as simple data flow or stream, in which data must be sent and retrieved at a periodic rate. The priority is to always get the newest data, with as little latency as possible. Data loss is acceptable to a certain extent, but (intermittent) communication failures can have severe consequences and will stop the robot from functioning. Due to potentially low periods, the data rates can be quite large which is a challenge for typical low-power link layers used in the IoT.

The communication between **levels 2 and 3** maps also to a data flow, but with potentially fewer requirements w.r.t. throughput and delay compared to the interface between levels 1 and 2. Indeed, data exchanges are generally expected to happen at a lower data rate, and a breakdown in communication can be handled by level 2.

The communication between **levels 3 and 4** characteristics are more diverse. For example, instead of continuously exchanging data about the robot's environment, level 3 can offer a service that follows the Request-response pattern. A typical examples of data exchanged by *levels 3 and 4* are trajectories, target positions, and high level state data. Contrary to the data flow communication of the levels below, this leads to a message type communication, shifting the requirements from being delay-dominated to being reliability-dominated. Due to significantly larger update periods the data rate requirements are also more relaxed.

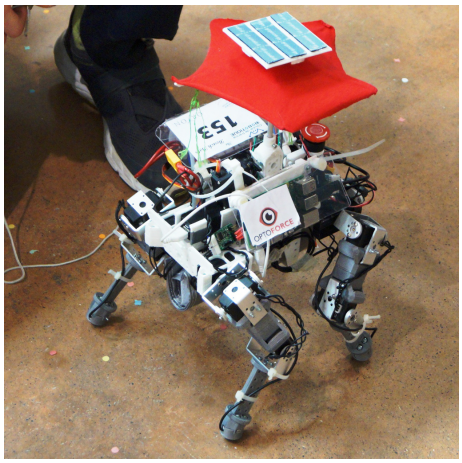

Figure 1. Buggybot, a four-legged IoT robot

In the IoT we see typically two types of communication patterns: Request-response and Publish-subscribe. With UDP being the predominant transport layer used when interacting with constrained devices, both of these patterns use a message type communication.

When mapping low-cost robots' communication needs with IoT communication patterns, it becomes apparent that the communication between *level 3* and *level 4* can be directly satisfied by typical IoT communication. Mapping the requirements of communication from *level 1 to 2* and from *level 2 to 3* is not quite as trivial. Since IP networks can make no guarantees for delays and data rates, the distributed software on these levels may suffer substantially (or even fatally) from jitter and data loss. The feasibility to distribute the software nodes on these levels thereby depends on the specific scenario - especially, on the targeted refresh periods, the used link layers, and control over the connecting network.

4.1 IoT Robotics communication using IoT technologies

In robotics, libraries like ROS or ZeroMQ [15] provide messaging infrastructures tailored exactly to the needs of robotics applications, providing the grounds for setting up low-delay data streams as well as reliable data exchanges. But as stated in section 3.1, these libraries need underlying OS support and are not designed for embedded environment, thus making it not feasible to run them on constrained devices.

The IoT domain has lately seen much standardization efforts, bringing the Internet protocol suite onto constrained low-power and low-cost hardware. The most prominent protocols defined in this space include IPv6 and 6LoWPAN [30], MQTT-SN [38], CoAP [37], and other application layer specifications such as DDS [32].

To mitigate the impact of unreliable IoT communication, two main approaches need to be explored (i) additional, smarter QoS mechanisms yet-to-be-defined should be used, and/or (ii) more resilient robotics algorithms could be used. Neither approaches are trivial, however. For example, though CoAP for example enables the use of both Request-response and Publish-subscribe patterns providing a reliable transport of data, it can not fully mitigate the shortcomings of the underlying IP network regarding potential problems with timing, jitter, and data rates.

5 Proof-of-Concept

As a proof of concept of convergence between low-cost robots hardware, software and communication, we modified a previously existing low-cost robot: **Buggybot** [23].

5.1 Initial Buggybot: HW & SW

Originally, Buggybot computation units were a Raspberry Pi and an Arduino Mega. The Arduino board handled low-level hardware e.g. sensors (infrared distance sensors, gyro, accelerometer) and actuators (servomotors enabling legs movement). The Pi handled Human-Robot interface (a web interface on Linux via TCP/IP over Wifi) and inverse kinematics computation (too heavy for the Arduino).

From the software perspective, Buggybot was initially divided in several "nodes": independent processes that communicate with one another to provide different services, extensively using the Aversive++ library and a custom hardware abstraction layer (HAL). The concept of Buggybot node was similar to a ROS node, but instead of ROS, we used ZeroMQ (ZMQ) to provide messaging, topic, publish/subscribe (and a custom protocol over serial to communicate with the Arduino). Seven types of Buggybot nodes composed the whole robotic application, grouped below by functionality levels :

- *Level 1 nodes:* The **embedded-io** node handles the low level hardware, and makes it available to others nodes. The **serial-controller** node manages the communication between the Arduino and the high-end computing unit (a Raspberry Pi, or a computer).
- *Level 2 nodes:* The **servo-mapper** node maps servomotors IDs to names, and convert angles in radians to servomotor's commands. The **Inverse Kinematics (ik)** node translates animations into angles for the servomotors. The **teleoperation-server** node plays an animation when a command is received.
- *Level and Level 4 nodes:* The **servo-mapper-ui** node is used to configure the servo-mapper node. The **teleoperation-client** node enables the user to send commands to the teleoperation-server.

5.2 Porting Buggybot to IoT

First, the Raspberry Pi and the Arduino Mega were replaced by a single IoT device: an Atmel SAMR21, based on a ARM Cortex M0+ microcontroller with 32kB of RAM and 256kB of flash memory. The SAMR21 board was chosen, because it is popular, off-the-shelf, has more than 2 UART ports and an IEEE 802.15.4 radio. Second, we ported Aversive++ to RIOT as described below, so that we could thus use RIOT's standard UDP/IPv6/6LoWPAN wireless communication stack instead of our custom protocols.

Offloading some Buggybot nodes' execution anywhere on the Internet hence became straightforward (we also used RIOT's border router implementation to connect the robot to the Internet). We then envisioned two modes of operation:

- **Config 1:** Level 1 nodes execute on the SAMR21 board, while other nodes are offloaded.

- **Config 2:** Both levels 1 and 2 execute on the SAMR21 board, and other nodes are offloaded.

In order to achieve **Config 1**, we had to modify two nodes: embedded-io, and serial-controller. Embedded-io's HAL was replaced by RIOT's hardware abstraction layer, and the serial controller node was transformed into a CoAP client that forwards (and translates) received ZMQ messages for the robot's 12 servomotors.

In order to achieve **Config 2** the biggest challenge was computing inverse kinematics fast enough on the SAMR21. This was made possible with an optimization using fixed point arithmetics, precomputed cosine tables, while lowering the precision.

5.3 Preliminary evaluation of Buggybot-IoT

For the robot to walk properly, the minimum refresh rate for the (12) servomotors is 20Hz. Hence, both Config 1 and Config 2 must process an animation frame in less than 50ms.

For **Config 1**, exchanged messages contain commands for 12 servomotors. Using the UDP/6LowPAN/CoAP networking stack, a message is 90 bytes long (24 bytes of data, 66 bytes of header). At a refresh rate of 20Hz, network throughput is thus 1800 B/s, which can be handled by IEEE 802.15.4 radios.

For **Config 2**, exchanged messages contain only the id of the animation to play. When a message is received by the robot, the animation is played until the next command is sent. A message is 67 bytes long (1 byte of data, 66 bytes of header). Considering on average of 1 command per second, network throughput is 67 B/s (much less than Config 1).

We first tested successfully **Config 1**, whereby Buggybot-IoT could walk as well as Buggybot. We measured a round trip time (between offloaded nodes and the robot) of 25 milliseconds, and an animation frame processing time on the robot of around 10ms, much less than the tolerated maximum (50ms). We concluded that the refresh rate could be increased, and that computing power is underused when the robot moves (around 20%) with Config 1.

We then tested **Config 2**, and measured that animation frame processing time on the robot is around 50ms in this configuration. Thus, while Config 2 requires much less radio throughput than Config 1, the former requires nearly 100% CPU usage when the robot is commanded to move at 20Hz.

6 Conclusion

In this paper we have highlighted the convergence of IoT hardware and low-cost robotics hardware. We have shown potential for further convergence in terms of embedded software platforms and network protocols for IoT robotics. Future work in terms of software platform include porting more comprehensive robotics libraries to RIOT, and the design of a ROS-like communication stack that works on low-end IoT devices. A key challenge will be achievable communication QoS guarantees for reliable remote execution in control loops.

7 References

[1] Arduino Mega. https://www.arduino.cc/en/Main/ArduinoBoardMega2560.

[2] Atmel SAMRR21 Xplained Pro. http://www.atmel.com/tools/ATSAMR21-XPRO.aspx.

[3] Aversive++ Library. http://aversiveplusplus.com.

[4] BBC:micro Project. https://www.microbit.co.uk/.

[5] Dtto - Explorer Modular Robot. https://hackaday.io/project/9976-dtto-explorer-modular-robot.

[6] Eistec Mulle Board. http://www.eistec.se/mulle/ .

[7] HikoB Fox. http://www.hikob.com/en/product/hikob-fox-mems-intertial-sensor/.

[8] IoT-lab Hardware: the IoT-lab M3 open node. https://www.iot-lab.info/hardware/m3/.

[9] Metabot is a DIY open-source legged robotics platform. http://metabot.cc.

[10] MyNewt Operating System. http://mynewt.apache.org.

[11] Open-source micro-robotic project. http://www.swarmrobot.org.

[12] Phytec phyNODE. https://github.com/RIOT-OS/RIOT/wiki/Board%3A-OpenMote.

[13] Phytec phyNODE. http://www.phytec.de/produkte/internet-of-things/phynode/.

[14] Zephyr Project Operating System. http://zephyrproject.org.

[15] ZeroMQ. http://zeromq.org.

[16] Zolertia Re-Mote board. http://zolertia.io/product/hardware/re-mote.

[17] The ARM mbed IoT Device Platform. 2016.

[18] E. Baccelli et al. RIOT OS: Towards an OS for the Internet of Things. In *IEEE INFOCOM*, 2013.

[19] C. Bormann et al. Terminology for constrained node networks. RFC 7228 (Informational), May 2014.

[20] M. Brambilla et al. Swarm robotics: a review from the swarm engineering perspective. *Swarm Intelligence*, 7(1):1–41, 2013.

[21] H. Bruyninckx. Open robot control software: the OROCOS project. In *IEEE ICRA*, 2001.

[22] M. Couceiro et al. A low-cost educational platform for swarm robotics. *International Journal of Robots, Education and Art*, 2(1):1–15, 2011.

[23] L. Dauphin. RIOT and Aversive++ experiment with Buggybot. https://github.com/xenomorphales/buggybot/tree/master/doc.

[24] R. Diankov and J. Kuffner. Openrave: A planning architecture for autonomous robotics. *Robotics Institute, Pittsburgh, PA, Tech. Rep. CMU-RI-TR-08-34*, 79, 2008.

[25] L. Grieco et al. IoT-aided robotics applications: Technological implications, target domains and open issues. *Computer Communications*, 54:32–47, 2014.

[26] O. Hahm et al. Operating Systems for Low-End Devices in the Internet of Things: a Survey. *IEEE Internet of Things Journal*, 2016.

[27] B. Kehoe et al. A Survey of Research on Cloud Robotics and Automation. *IEEE Trans on Automation Science and Engineering*, 2015.

[28] G. A. Korash et al. African Robotics Network and the 10 Dollar Robot Design Challenge. *IEEE Mag. on Robotics & Automation*, 2013.

[29] M. Lapeyre et al. Poppy: Open Source 3D-printed Robot for Experiments in Developmental Robotics. In *IEEE ICDL*, 2014.

[30] K. N. et al. IPv6 over low-power wireless personal area networks (6LoWPANs): overview, assumptions, problem statement, and goals. Technical report, 2007.

[31] S. Noirpoudre. Dans la famille Poppy, je voudrais? le robot Ergo Jr !, 2016.

[32] G. Pardo-Castellote. OMG Data Distribution Service: Architectural Overview. In *Distributed Computing Systems Workshops, 2003. Proceedings. 23rd International Conference on*, pages 200–206. IEEE, 2003.

[33] H. Petersen et al. IoT Meets Robotics-First Steps, RIOT Car, and Perspectives. In *ACM International Conference on Embedded Wireless Systems and Networks (EWSN)*, 2016.

[34] M. Quigley et al. ROS: an Open-Source Robot Operating System. In *ICRA Workshop on Open-Source Software*, 2009.

[35] M. Rubenstein. Programmable self-assembly in a thousand-robot swarm. *Science*, 345:795–799.

[36] M. Satoshi et al. M-tran: Self-reconfigurable modular robotic system. *IEEE/ASME Transactions on Mechatronics*, 2002.

[37] Z. Shelby et al. The constrained application protocol (coap). Technical report, 2014.

[38] A. Stanford-Clark and H. L. Truong. Mqtt for sensor networks protocol specification. *International Business Machines Corporation version*, 1, 2008.