J. Sens. Actuator Netw. **2013**, *2*, 288-315; doi:10.3390/jsan2020288

OPEN ACCESS

Journal of Sensor and Actuator Networks

ISSN 2224-2708

www.mdpi.com/journal/jsan/

Article

TinyCoAP: A Novel Constrained Application Protocol (CoAP) Implementation for Embedding RESTful Web Services in Wireless Sensor Networks Based on TinyOS

Alessandro Ludovici [1,2], Pol Moreno [1,2] and Anna Calveras [1,2,*]

[1] Wireless Network Group (WNG), Department of Telematics Engineering, Universitat Politècnica de Catalunya, C/Jordi Girona 1-3, Mòdul C3, Barcelona 08034, Spain

[2] i2CAT Foundation, Gran Capità 2-4 (Nexus Building), 08034 Barcelona, Spain; E-Mails: alessandro.ludovici@entel.upc.edu.com (A.L.); pol.moreno@entel.upc.edu (P.M.)

* Author to whom correspondence should be addressed; E-Mail: anna.calveras@entel.upc.edu; Tel.: +34-93-401-6013; Fax: +34-93-401-1058.

Received: 12 March 2013; in revised form: 16 April 2013 / Accepted: 26 April 2013 / Published: 14 May 2013

Abstract: In this paper we present the design and implementation of the Constrained Application Protocol (CoAP) for TinyOS, which we refer to as TinyCoAP. CoAP seeks to apply the same application transfer paradigm and basic features of HTTP to constrained networks, while maintaining a simple design and low overhead. The design constraints of Wireless Sensor Networks (WSNs) require special attention in the design process of the CoAP implementation. We argue that better performance and minimal resource consumption can be achieved developing a native library for the operating system embedded in the network. TinyOS already includes in its distribution an implementation of CoAP called CoapBlip. However, this is based on a library not originally designed to meet the requirements of TinyOS. We demonstrate the effectiveness of our approach by a comprehensive performance evaluation. In particular, we test and evaluate TinyCoAP and CoapBlip in a real scenario, as well as solutions based on HTTP. The evaluation is performed in terms of latency, memory occupation, and energy consumption. Furthermore, we evaluate the reliability of each solution by measuring the goodput obtained in a channel affected by Rayleigh fading. We also include a study on the effects that high workloads have on a server.

Keywords: CoAP; REST; WSNs; 6LoWPAN; HTTP; TinyOS

1. Introduction

The interest of industries and researchers in the application of the Internet communication paradigm in constrained networks and devices has grown to become one of the most promising developments in the Internet of the future. In particular, a great deal of effort has been focused on the integration of Wireless Sensor Networks (WSNs) and the Internet.

The main interest in making WSNs part of the Internet is to allow both to interact with each other using the existing Web technologies. Therefore, the information provided by sensors could be distributed through standard Web mechanisms, and sensors would be considered like any other Web resource. From this perspective, WSNs would no longer be stand-alone networks but part of ubiquitous networks. We refer to this new approach as the Web of Things (WoT).

Enabling IP in WSNs is the first step towards the realization of the WoT. In this sense, the IPv6 over Low power Personal Area Networks (6LoWPANs) [1] protocol is an example of a lightweight IP implementation for WSNs. 6LoWPAN enables the transmission of IPv6 packets in networks adopting the IEEE 802.15.4 standard [2]. An IETF specific work group has detailed its definition in RFC 6282 [3]. 6LoWPAN would only provide IP connectivity but not interoperability at higher layers. In this regard, the use of Web services has been proposed [4,5].

Web services can be developed according to the REST [6] or the SOAP [7] architectural styles. Web services complying with the REST principles are called RESTful. Many research papers have analyzed the impact that both architectures would have on WSNs. Results from [8] and [9] show that the overhead of the messages exchanged between SOAP Web services is higher than that of RESTful Web services. In fact, SOAP messages are encapsulated in the XML body, which involves a complex and expensive message processing. However, it is possible for the REST data formatting to be based on different languages, which enables the overhead to be kept low. The authors of [9] suggest that the use of SOAP should be avoided, and consider employing RESTful Web services instead. An early work considering the application of REST principles in WSNs is presented in [4]. However, it does not take into account the use of Web services to access and interact with WSN nodes, but only for discovery of devices.

An IETF work group called Constrained RESTful Environments (CoRE) [10] has been created with the aim of contributing to the development and standardization of RESTful Web services for constrained networks. With this approach, the work group defined a new Web transfer protocol called Constrained Application Protocol (CoAP) [11]. CoAP seeks to apply the same application transfer paradigm and basic features of HTTP to constrained networks, while maintaining a simple design and low overhead. Unlike HTTP, CoAP uses UDP as transport protocol. This choice would enable CoAP to have a low impact on the limited bandwidth of the 802.15.4 wireless links. However, since UDP is an unreliable protocol, CoAP has to implement its own mechanisms in order to guarantee reliability to those applications that use it.

Recently, many implementations of CoAP have appeared. These are Californium [12], jCoAP [13], CoAPy [14] and Libcoap [15]. However, they are not related to any WSN operating system (OS) or platform. They provide libraries and APIs written in different languages that can be used for embedding CoAP in WSN nodes or for developing applications in other environments. The use of these libraries in WSNs, however, requires that they must be adapted to the OS in use. The adaptation process, however, could not guarantee adequate performance or reliable execution. Considering the resource constraints of typical WSN nodes, a different approach is required. This should consider the development of a native implementation for the OS used in the WSN. This approach could yield better performance, less resource consumption and more reliable software. However, more powerful networks and devices could still use complex libraries or APIs such as those listed above.

Typical WSNs nodes are battery-powered and often deployed in unattended environment, therefore software applications should be optimized concerning lowest energy consumption and reliable execution. The reduction of energy consumption is mainly achieved using radio duty cycling protocols. Further reduction can be reached through effective memory management. This allows saving CPU cycles and reducing the code complexity. Both aspects are critical for lowering the energy consumed by the CPU processing and for reducing the risks of failures during execution. The optimization process of our CoAP implementation is focused in developing an efficient and safe use of memory. The results obtained in our tests prove the importance that memory management has on performance and resource consumption. This paper focuses on severely constrained WSNs nodes featuring few kilobytes of memory and CPUs with reduced computational capabilities. Although some actual technology offers more powerful nodes, working with these constrained nodes help us to fix a lower bound to the resources that could be used by our implementation. In that way, we ensure that our design choices can deliver a highly optimized implementation suitable for any application domain.

CoAP has been already implemented in the most popular OSs for WSNs such as Contiki [16] and TinyOS [15,17]. In particular, an implementation based on the Libcoap library has been released in the latest distribution of TinyOS. We refer to this implementation as CoapBlip. However, we detected some weaknesses in its design that may have a negative impact on the resource consumption and functioning of sensor nodes. Furthermore, being an adaptation of an external library its code does not result optimized for TinyOS. The implementation presented in [17] has been developed on top of a 6LoWPAN implementation that is no longer supported, and is affected by various limitations [18,19]. Therefore, the existing CoAP implementations for TinyOS would not allow either the functionalities or the performance offered by this protocol to be fully realized. To overcome this problem, we design and implement our original implementation of CoAP for TinyOS, which we refer to as TinyCoAP (http://sourceforge.net/projects/tinycoap/).

The CoAP definition developed by the CoRE work group is still under standardization, and its use in WSNs has been studied in several works [15–17,20–22]. However, none of these papers contain an extensive evaluation of CoAP. In particular, the reliability mechanism provided by the protocol has not yet been evaluated. Although a comparison between CoAP and HTTP has been made in [22], it does not consider the possibility of using HTTP with transport protocols different from TCP. In our tests, we evaluate the performance of an HTTP server using UDP and persistent TCP connections. In fact, a fair comparison between CoAP and HTTP should at least include the above-mentioned possibilities in order to minimize the effects that TCP has on the HTTP performance, otherwise a comparison between CoAP and HTTP would result in an evaluation of UDP and TCP.

In this paper, we provide several original contributions. First, we present and evaluate our original implementation of a CoAP library native for TinyOS. Then, we provide a comprehensive analysis of the functioning of CoAP including an evaluation of the reliability mechanism. The analysis is conducted by carrying out an extensive evaluation of CoAP performance in a real WSN. In particular, we compare the performance of TinyCoAP with those of HTTP and CoapBlip. The evaluation of each solution is performed in terms of energy, latency, and memory occupation. The effects that high request rates have on performance have been also evaluated. Moreover, we test the reliability mechanisms provided by each solution, considering an 802.15.4 channel under Rayleigh fading. To the best of our knowledge, this is the first work in which the behavior of the reliability mechanism of CoAP has been studied.

The paper is organized as follows. First, in the following section we present the definition of CoAP and other existing proposals of application protocols for WSNs. Then, in Section 3, we present the related work. The implementation of TinyCoAP is illustrated in Section 4. The test-bed used for our experiments is described in Section 5. Results and discussion of the performance evaluation are reported in Section 6. Finally, in Section 7 we conclude the paper and give guidelines for future work.

2. Constrained Application Protocol

In this section we present the state-of-the-art of application protocols for WSNs. We focus on the protocols that provide an alternative to HTTP for building RESTful Web services. Then, we introduce the CoAP protocol.

2.1. Application Protocols for WSNs

The HTTP/TCP stack is the standard solution used for building RESTful Web services. The implementations presented in [23–28] are the first examples that proved the feasibility of applying the HTTP/TCP stack in constrained networks. However, these solutions are tailored for specific applications. As a consequence, only the TCP or IP capabilities that are needed by the application are implemented. Many of these implementations miss of IP fragmentation, retransmission or congestion control mechanisms.

The authors of [29] have analyzed the resource costs of building Web services for WSNs following the HTTP/TCP paradigm. The authors suggest design options that could lower the resource consumption. The use of persistent TCP connections as well as the avoidance of delayed acknowledgments has been proposed. The use of HTTP 1.0 is also suggested for its better integration with the persistent TCP connections. In our paper we evaluated an HTTP embedded Web Server adopting these design principles.

The latest research trends have considered the study of alternative solutions to the HTTP/TCP stack. Although feasible, the application of this stack in constrained networks presents several issues regarding its overhead and impact on bandwidth. In fact, the HTTP protocol header is too chatty for the limited space available in 6LoWPAN frames. Furthermore, as reported in [30], HTTP has evolved into a complex protocol involving optional headers and a number of features that increase the complexity of its implementation in constrained devices. Finally, the TCP protocol would have a poor effect on the available bandwidth of WSNs. However, a HTTP/TCP solution would ensure interoperability between different networks without the need for any intermediate translation proxy. The adoption of a lightweight binary protocol that could be easily mapped to HTTP would meet both requirements.

An early proposal for a protocol to interact with sensors using RESTful Web services is presented in [5]. This protocol is called TinyREST. The TinyREST commands were composed by combining the request method with the URI of the sensor resource. These request methods were limited to the POST, GET and SUBSCRIBE requests of HTTP. Instead of using IP, TinyREST uses the networking tools provided by the TinyOS active message stack. The messages were transmitted as plain ASCII and generated an overhead of 29 bytes. The use of TinyREST was intended only for communications inside the WSN. The interaction with external Web services was performed through a gateway responsible for translating the HTTP messages into TinyREST messages.

A proposal for a binary version of HTTP, which is called Embedded Binary HTTP (EBHTTP), has been defined in [31]. An application of EBHTTP for building RESTful Web Services has been considered in [32]. EBHTTP is a binary-formatted and stateless encoding of the standard HTTP protocol. Its use is intended for resource constrained WSNs. The design of this protocol focuses in reducing the overhead of HTTP while maintaining the same HTTP semantics and communication paradigm. EBHTTP uses the UDP protocol instead of TCP. The first proposal of the CoRE working group, called Chopan [33], follows the EBHTTP idea of compressing HTTP messages into a binary format. As EBHTTP, Chopan uses UDP as transport protocol. The proposal also included the presence of transparent caching and gateways for translating Chopan into HTTP. The weaknesses of the Chopan and EBHTTP proposals consist of the lack of reliability and in the processing power required to encode the HTTP protocol. In fact, although the processing required to encode the protocol would be less than that required by a direct use of HTTP, it would be still expensive due to the limited resource of a sensor. Furthermore, although they are bound on UDP, they do not provide any reliability mechanism. As a result of these limitations, attention has been shifted to the definition of CoAP. Next, we detail the definition and functioning of CoAP.

2.2. CoAP Definition

From a theoretical perspective, CoAP can be regarded as a two-layer protocol [11]. A message layer is used to deal with UDP, while the other layer is used for request/response interactions using methods and response codes, as done in HTTP. In fact, in contrast to the interaction model of HTTP, CoAP exchanges messages asynchronously. Thus, by splitting CoAP in two layers, request/response interactions would be transparent to the asynchronous nature of the message layer. Figure 1 shows the conceptual layering of CoAP.

Reliability is provided by the message layer and is activated when Confirmable (CON) messages are used. It is implemented through end-to-end stop-and-wait retransmissions of requests with exponential backoff [11]. Furthermore, CoAP is able to match requests to responses so that it can detect if a message is duplicated. This is possible because messages are labeled with an identification number generated by the client and inserted into the CoAP header. Finally, a token is used for differentiating between concurrent requests. CoAP also defines Non-confirmable (NON) messages to allow sending requests that does not require reliability. The detection of duplicated messages is available in CON as well as in NON messages.

Figure 1. Layering of Constrained Application Protocol (CoAP).

```
+----------------------+
|      Application     |
+----------------------+
+----------------------+
|  Requests/Responses  |
|----------------------|  CoAP
|       Messages       |
+----------------------+
+----------------------+
|          UDP         |
+----------------------+
```

A server receiving a CON request must acknowledge its receipt to the client that initiated the communication. The server might send an empty ACK to indicate that the response will be deferred. In this case, the client must acknowledge the arrival of the CON response message. Should the response be immediate, the ACK will contain the response and the transaction will end with the reception of the ACK. After sending a CON request message, the client starts a timeout with exponential backoff in order to retransmit the request periodically in case it has not been acknowledged. Finally, a server might send a RST response to indicate that it is not able to process the CON request. NON messages are used when reliability is not required.

Request methods are constrained to GET, POST, PUT and DELETE methods of HTTP. As regards the response codes, CoAP defines its own codes and also uses a small subset of the HTTP codes.

The CoAP message is composed of a header, options and a payload; the header has a length fixed to five bytes, while the options may have a variable length. Figure 2 shows the CoAP message format.

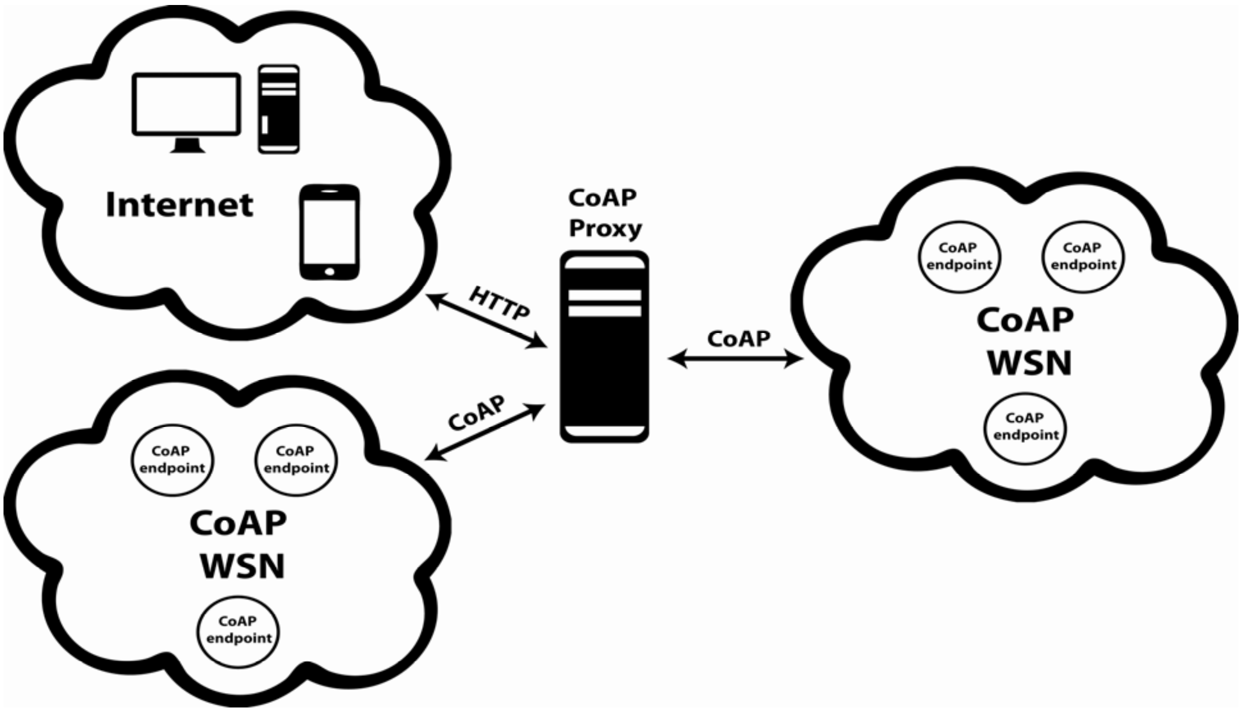
Figure 2. Message format.

```
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  TKL  |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Token (if any, TKL bytes) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|1 1 1 1 1 1 1 1|    Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3 shows the architecture of a CoAP-based WSN. According to [11], a CoAP WSN node can have the function of both client and server simultaneously. The term endpoint is used to refer to a CoAP node. However, in this paper we will refer to it as a server. This allows us to give greater emphasis to the function that the CoAP node has in our experiments and thus avoid confusion. A

CoAP proxy can be used to enable the communication between a CoAP based WSN and a HTTP external network. This proxy can work also as a gateway for connecting to other WSNs. The implementation of the proxy is out of the scope of this paper.

Figure 3. Architecture of a CoAP-based Wireless Sensor Network (WSN). The proxy enables integration between the WSN and external networks that use HTTP**.**



CoAP is particularly suitable for applications based on machine-to-machine communications. As an example, CoAP could be applied in smart energy applications to provide end-to-end connectivity to energy providers and the equipment of consumers.

3. Related Works

As previously anticipated, various studies have discussed the application of CoAP in WSNs. In this section we give details about these works.

Previous work by the authors [34] motivated the choice of the REST architecture and the adoption of the CoAP protocol. We also suggest modifications to an early format of the protocol and discuss possible problems of its implementation.

The authors of [16] presented a CoAP implementation for Contiki. The aim of this implementation was to achieve high-energy efficiency by leveraging a radio duty cycling mechanism. The implementation has been evaluated in a multi-hop network. The collected results have shown that energy consumption is lower when using a radio duty cycle but leads to in a worsening of the latency performance.

In [20], the authors report a simple comparison of CoAP and HTTP in terms of energy consumption. This work also describes the design of a gateway used to connect a CoAP based WSN to an external IP network that uses HTTP. In [22], the authors of [20] compared the performance of CoAP to that of HTTP. The evaluation was carried out on the basis of energy consumption and response time. In particular, energy consumption was evaluated by means of simulation. The response time was measured in a real WSN. Both experiments were conducted considering a client querying an embedded server to obtain temperature and humidity values. The energy consumed was measured according to the variation of the interarrival packet time. The response time was calculated for the case where the server was at a distance of 1-hop and 2-hop from the client. The results obtained show that CoAP yields a better performance in both the evaluation parameters considered by the authors.

A study on network sensor deployment [21] used CoAP and HTTP as data transport protocol for sensor network reprogramming. Both protocols were evaluated over a duty cycled radio layer. Results were obtained through simulation and show that CoAP and HTTP provide similar results. In [35], the authors present a framework for M2M communications using CoAP. They also present an improved publish/subscribe mechanism also based in CoAP. Both solutions are evaluated showing the advantage of using CoAP instead of HTTP.

The authors of [36] give an overview of the current CoAP implementations and present the results of an interoperability meeting organized by the European Telecommunications Standards Institute (ETSI). In [17], the authors present a CoAP implementation for TinyOS and the implementation of a compression mechanism of the XML format. A performance evaluation was carried out considering the CoAP request success probability as a function of the request rate of the client node. Furthermore, the authors report results from an evaluation of the memory occupation of the TinyOS components used in their implementation. Finally, the authors proved the efficiency of the XML compression scheme by studying its processing-time. As previously commented, this CoAP implementation was developed on top of an unsupported and limited 6LoWPAN implementation named 6lowpancli [37]. In particular, as pointed out in [18,19], 6lowpancli provides only basic functionalities of 6LoWPAN. 6Lowpancli doesn’t support any type of neighbor discovery mechanism, it is completely static and requires manual configuration. As reported in [18,19] the support for mesh network is not provided and when a packet with different destination address is received, it is just dropped. The results of a performance evaluation done in [18] show that 6lowpancli does not perform well in terms of energy consumption and latency. Thereby, its limitation would affect any implementation build on top of it.

As previously mentioned, CoapBlip is currently included in the latest distribution of TinyOS. The authors of CoapBlip present its design in [15]. They evaluated their implementation and compared it to HTTP. The performance evaluation considers the ROM occupation and the average response time of CoAP and HTTP. Preliminary results of an evaluation show that CoAP yields better performance than HTTP. In [38], CoapBlip has been used to evaluate the CoAP protocol in combination with other low layer protocols. In this sense, it has been evaluated along with the Routing Protocol for Low-power and Lossy Networks (RPL) and the Low Power Listening (LPL) protocol. In the next section we give details about our original CoAP implementation for TinyOS.

4. Implementation

One critical WSN design challenge involves providing reliable and performing solutions while coping with constrained resources. Memory, energy and bandwidth represent the resource constraints to meet. Thereby, the design process at any level has to be focused on optimizing their use.

In this paper, we develop a CoAP library for TinyOS to be embedded in highly constrained WSNs. TinyCoAP implements all the features of CoAP [11] except for the blockwise transfer. The goal of our implementation is to minimize the resource consumption by developing a lightweight and efficient code optimized for the OS in use. Because of its popularity and diffusion, TinyOS has been chosen as the OS target.

TinyOS [39] is an embedded OS for WSNs designed to meet the requirements of constrained networks and devices. It is composed by a set of reusable components that can be used to build specific applications. TinyOS is implemented in the NesC language [40]. NesC is a C dialect designed to improve code efficiency and robustness in embedded software applications [39]. Through its simplicity, NesC is able to reduce RAM occupation, code size, and prevents low-level bugs. The programming model of TinyOS is also based on this language.

Besides NesC, TinyOS allows using more complex languages such as Java, Python or C. In particular, C code can be embedded in nesC programs or it can be used to build libraries that TinyOS components can use. As we will explain later in this section, a TinyOS based WSN can achieve better performance and be more reliable when using exclusively NesC.

The design philosophy of TinyCoAP follows the principles of the TinyOS programming model. The code is structured in TinyOS components and the use of external libraries is avoided. TinyCoAP is completely written in the NesC language. The rest of this section focuses on the memory allocation system, library and the data structures of TinyCoAP.

4.1. Structure of the Library

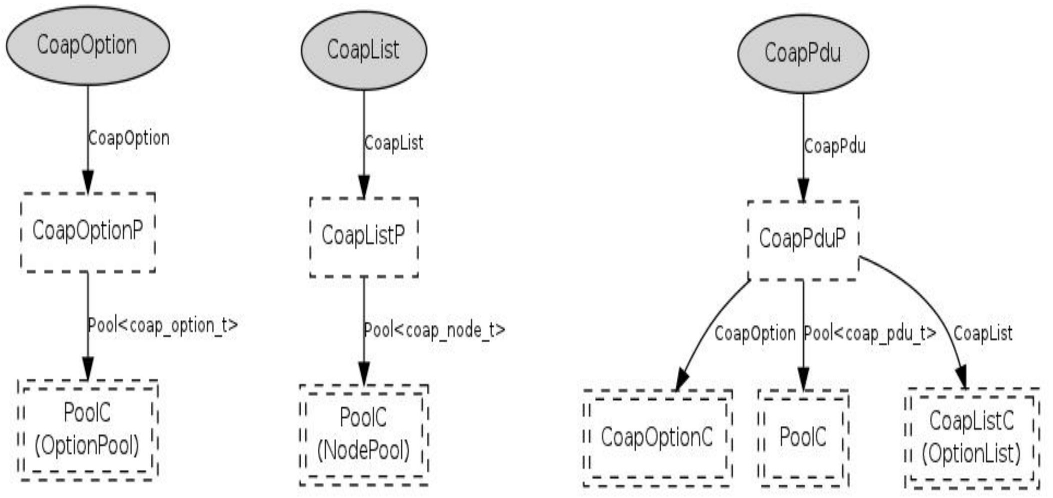
TinyCoAP provides a CoAP library native for TinyOS. It is designed behind the idea that better performance and reliable run-time execution are both achieved integrating it with the OS core libraries. Following these design principles, the core functionalities of CoAP are provided as TinyOS components. These components are developed as part of the TinyOS network library. Differently from TinyCoAP, CoapBlip is thought as an adaptation of a C library for generic embedded systems. A TinyOS component is used as an adapter between this library and the TinyOS application.

TinyCoAP avoids using external C libraries and relies completely on code developed in the NesC language. This allows reaching a high code optimization and having less impact on the WSN node memory. These benefits derive mainly from the different organization and functioning of C and NesC programs. Typical C programs are composed by functions that are specified in separated files. These are compiled separately and then linked together by matching global name of functions. The interaction between functions is achieved dynamically during run-time by using function pointers. Pointers are stored in the RAM memory and therefore cause a growth of its occupancy. In contrast with C, TinyOS programs are conceived as a set of components connected together to perform a specific task. The interaction between components is done using the interfaces that these provide. Applications declare at compile-time which components they use and then, they explicitly wire the interfaces they will use at run-time. Thanks to this static wiring, TinyOS programs avoid using function pointers and therefore they are able to reduce the RAM memory footprint.

The TinyCoAP library is composed by five components. Its design follows the CoAP conceptual layering. The message layer is implemented by three components. CoapPDU, where PDU stands for

Protocol Data Unit, is the main component of this sub-layer. It provides the interface used to create, read and write CoAP packets. The interface needed to create or delete options is provided by the CoapOption component. The creation, use and managing of the linked lists is performed by the interface provided by the CoapList component. Linked lists are useful for iterating the packets that are in the memory pool waiting for being processed. CoapList is also used to store and iterate the options that compose a packet and to manage retransmissions. CoapPDU is wired to CoapList and CoapOption. This allows CoapPDU to work with the options contained in a CoAP packet. Moreover, each component of the message layer is wired to the TinyOS PoolC component. This is used to allocate the memory needed to perform their operation. PoolC allocates memory according to the data structure that is specified by each component. The wiring of the message layer components is shown in Figure 4.

Figure 4. Wiring of the TinyCoAP interface for the CoAP message layer. PoolC is used to provide the memory needed by the components.



The request/response matching layer of CoAP is implemented by the CoapServer and CoapClient components. The CoapClient component provides the interface used to send CoAP requests. The interface provided by CoapServer allows initializing and binding the server to a specific UDP port. The management of the retransmission mechanism and the CoAP packet processing are also implemented in these components. The CoapServer component implement the discovery of CoAP resources [41] and the observe model of CoAP [42]. The management of the resources provided by the server is implemented in a distinct interface. The resources are created through a parameterized interface. This is called CoapResource and provides commands and events to handle resources and the separate response mechanism of CoAP.

4.2. RAM Memory Allocation

Managing the allocation of RAM memory is one of the most critical aspects to consider when embedding software applications in WSN nodes. The management of memory allocation has to cope with the limited size of RAM memory and the lack of hardware memory protection that characterize constrained nodes. In this perspective, managing the RAM memory dynamically could increase the probability of having failure nodes or could exhaust the available memory. In fact, the lack of hardware memory protection does not prevent the risks of having a collision between the heap and stack or a memory leak [41]. Furthermore, the size of the allocated RAM memory would be difficult to control with this allocation system.

TinyCoAP avoids these risks by allocating RAM memory statically. The size of the allocated memory is known at compile time and the possibility of memory exhaustion is therefore avoided. Furthermore, static allocation would eliminate the risks of failures due to collision of the heap and the stack. Therefore it would enhance the network reliability. A further optimization is obtained allowing to TinyCoAP to create CoAP responses without allocating new memory. In TinyCoAP the responses are created using the memory already allocated for storing the relative CoAP requests. Besides the reduction of the RAM memory footprint this allows enabling a lighter packet processing with less impact on the CPU. As a consequence, the reduction of the CPU use would lower the energy consumption. As reported in [43], the CPU consumes 4.6 mA when active and 2.4 mA when idle while the radio uses 3.9 mA when receiving. Therefore, the TinyCoAP management of buffers would save CPU cycles and enhance the battery life of nodes.

The static allocation of memory done by TinyCoAP is compliant with the RAM memory management defined in NesC. In fact, NesC does not support dynamic memory allocation. This characteristic allows preventing memory fragmentation and run-time allocation failures [39]. However, a situation may arise in which applications might need dynamic allocation. To overcome this problem, TinyOS provides a component called PoolC that simulates the dynamic memory allocation. Should PoolC be used, the maximum pool memory size would be allocated statically at compile time. During the execution time, the applications will take the amount of RAM memory they need from that available in the pool. An eventual memory leak would cause the pool to empty, but the heap and stack would not collide. As mentioned above, TinyCoAP uses PoolC to allocate the buffers needed to store the CoAP packets and the linked lists. Differently from TinyCoAP, CoapBlip adopts a dynamic memory allocation management. It uses the malloc memory management library to allocate memory for buffers and linked lists.

4.3. Data Structure

As mentioned above, the TinyCoAP components are organized following the conceptual layering of CoAP. The message layer is build on top of the 6LoWPAN implementation distributed in TinyOS named Blip [44]. CoapBlip also uses this 6LoWPAN stack. Should Blip receive a UDP packet, it checks the presence of the CoAP header. If this header is present, the interface provided by CoapPDU saves it in a CoAP PDU. This PDU is stored in the memory previously allocated through PoolC. The use of PoolC allows TinyCoAP to establish at compile time the maximum size a packet can have and the maximum number of packets it can handle. The maximum length of options and the maximum number of packets that can be queued by a node can also be specified. These features make TinyCoAP robust against possible memory leaks and always provide it with room in the memory for the incoming packets. Furthermore, TinyCoAP is easily adaptable to the requirements of different applications. The TinyCoAP PDU data structure is designed it to be used with PoolC. It avoids the use of pointers for accessing to the different parts of the PDU. Table 1 shows the CoAP PDU defined in CoapBlip and TinyCoAP.

Table 1. CoAP PDU structures. CoapBlip stores the PDU in the UDP buffer and uses a pointer to provide access. TinyCoAP saves it in the memory allocated with PoolC.

CoapBlip	TinyCoAP
typedef struct {	typedef struct {
coap_hdr_t *hdr;	uint8_t timestamp;
unsigned short length;	coap_hdr_t hdr;
coap_list_t *options;	struct sockaddr_in6 addr;
unsigned char *data;	uint8_t payload [MAX];
} coap_pdu_t;	uint16_t payload_len;
	coap_list_t opt_list;
	} coap_pdu_t;

In TinyCoAP, the received CoAP message is initially stored in the UDP buffer as a void element. This element is then converted into a coap_pdu_t structure and stored in the memory pool. Once the PDU structure has been created, the UDP buffer is ready to receive a new incoming packet. TinyCoAP enables the maximum payload allowed for requests and responses to be defined at compile time. Thus, the memory usage can be adjusted to the application requirements and to the particular use of the sensor.

CoapBlip uses pointers for accessing the different parts of the PDU. Should a CoAP packet be received, CoapBlip stores it in a buffer allocated through malloc and initializes the pointers defined in coap_pdu_t. This buffer is placed at UDP level and its size is always equal to the maximum packet size allowed by CoapBlip. Thus, although CoapBlip uses malloc, the memory is always allocated with the same size.

5. Test-Bed

As previously mentioned, in this paper we compare and discuss the performance obtained in a real 6LoWPAN network by TinyCoAP, CoapBlip and HTTP. Our experiments involve different solutions for the transport layer used by HTTP. We consider HTTP/TCP, HTTP/UDP and HTTP persistent. The third solution refers to the use of a persistent TCP connection.

The HTTP and TCP protocols that we use to implement the server are included in Blip. The HTTP version it implements is the 1.0. The TCP version used by the client is one the most widely adopted, which is the TCP Reno. The congestion window size (CWND) of client and server is fixed to 1. A larger CWND could overload the wireless link with too many transmissions, and thereby increasing the probability of collision. The need to have more than one TCP packet pending in the network could be justified in a multi-hop scenario. In a one-hop network there is not such a need, and client and server would avoid competing for the radio channel. The MSS of both sides always corresponds to the length of the TCP payload sent in each experiment. Thus, each HTTP single packet is sent in one TCP segment. This setting allows maximizing the throughput by reducing the interchange of control messages and avoiding fragmentation.

The tests involve client/server transactions where a client sends requests to a server in order to retrieve information. All the requests are sent using the GET method. When receiving a request with test as URI, the CoAP or HTTP server replies with a payload composed of a sequence of bits of fixed size. In this way, the mote performs no sensing operation that might influence the results. Therefore, the experiments account only for the performance of each technique in processing and replying to the received messages.

The CoAP CON request messages sent by the client have a total length of 14 bytes. The HTTP GET requests carry the same information of that sent in CoAP. However, the verbose format of HTTP implies a growth in size of up to 37 bytes. Table 2 shows the composition of the GET requests for CoAP and HTTP.

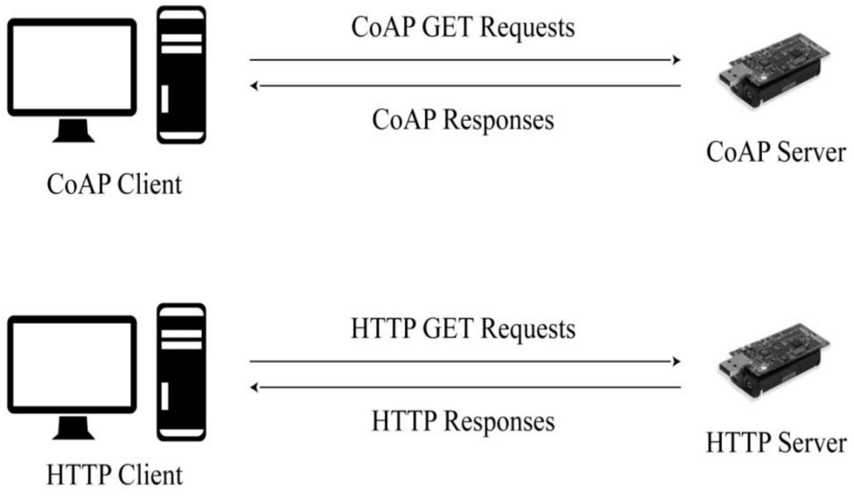
Table 2. Composition of the HTTP and CoAP Confirmable (CON) requests. In both cases the requests are sent using the GET method.

CoAP	HTTP
CoAP HEADER: 5 bytes	*GET /test HTTP/1.0\r\n*
COAP URI_PATH: 6 bytes	*Host: fec0::2\r\n\r\n*
COAP TOKEN: 3 bytes	
PAYLOAD: 0 bytes	

All the tests are performed in a real WSN implementation. Crossbow's TelosB motes are used as a hardware platform to develop our test-bed network. TelosB [45] is a typical example of a low-cost wireless sensor used in constrained sensor networks. It features 16-bit RISC MCU at 8 MHz and 16 registers. The platform offers 10 kB of RAM, 48 kB of flash memory and 16 kB of EEPROM.

Since our experiments are focused on evaluating single client/server transactions, we can keep our test-bed network simple and avoid deploying complex architectures. The test-bed network used in all the experiments is shown in Figure 5. The CoAP and HTTP clients as well as the proxy server are located in a PC. Each request is sent through a 6LoWPAN base station attached to the USB port of the PC. The server is embedded in a TelosB mote located one-hop away from the 6LoWPAN base station.

Figure 5. Test-bed network. The HTTP or CoAP clients are located in a PC while the servers are embedded in a sensor.



6. Results

In this section, we report the performance evaluation for all the considered solutions. Our study evaluates various parameters. First, we measure the amount of RAM and ROM memory used by each solution; we then evaluate the latency of request/response transactions; after that, we measure the energy consumed by each different solution for processing and replying to a request. Moreover, we evaluate the client goodput obtained in 802.15.4 links affected by Rayleigh fading. This test allows us to evaluate the performance of the reliability mechanism proposed for CoAP [11]. Finally, we evaluate the effect that high workloads have on the server performance. In this case, we analyze the rate of the requests per second that it can serve as a function of the client request rate.

The results concerning latency, energy and reliability are reported according to the payload size with which a server replies to a client request. The maximum payload size varies depending to the implementation used. We found the cause of this variation on the different methods used to allocate RAM memory. The implementations using dynamic memory allocation achieve the lowest payload size. In this sense, CoapBlip is able to reach 650 bytes as maximum payload size while HTTP/TCP and HTTP persistent reach 800 bytes. TinyCoAP benefits from the use of static allocation and it is able to reach 1,200 bytes. The same size is achieved by HTTP/UDP. In this case the low complexity of the implementation and the absence of HTTP buffers allow to HTTP to work with high payload sizes.

6.1. Memory Occupation

Table 3 shows the amount of RAM and ROM memory allocated at compile time for each considered implementation. The values correspond to a maximum payload size of 500 bytes. The values for HTTP/TCP and HTTP persistent are the same and are reported as HTTP. In fact, both solutions use the same TCP buffers and the use of a persistent connection does not vary their allocation.

Table 3. RAM and ROM memory occupation. TinyCoAP reserves all the memory required at compile time.

	TinyCoAP	**CoapBlip**	**HTTP**	**HTTP/UDP**
RAM	8,458 kB	7,102 kB	7,85 kB	3,922 kB
ROM	31,812 kB	42,576 kB	39,484 kB	27,802 kB

TinyCoAP occupies more RAM memory than the other solutions, since it allocates all the memory needed for buffering the CoAP packets at compile time. Therefore, the occupation of RAM would remain at the same level during the functioning of the node while those of the other implementation would increase.

The occupation of ROM memory gives an idea of the complexity and weight of the code of each implementation. In fact, it is in the ROM memory where the compiled code will be stored. A code with a small memory footprint would allow adding further functionality such as resources that the server could provide to clients or enrich CoAP with more capabilities.

CoapBlip has the highest ROM memory footprint. We found the main cause in the lack of optimization of the code. As mentioned in Section 4, CoapBlip is an adaptation of a C library. This library is therefore installed in the node along with the TinyOS component used for its adaptation to this OS. The use of C libraries is usually too complex for the memory constraints of a mote and implies a growth of the memory footprint. Also HTTP solutions using TCP make use of a C library, thereby the ROM footprint increase also for these implementations. TinyCoAP lowers the ROM footprint, thus avoiding the use of C libraries, and consists only of a code written in the TinyOS nesC language and therefore optimized for this environment.

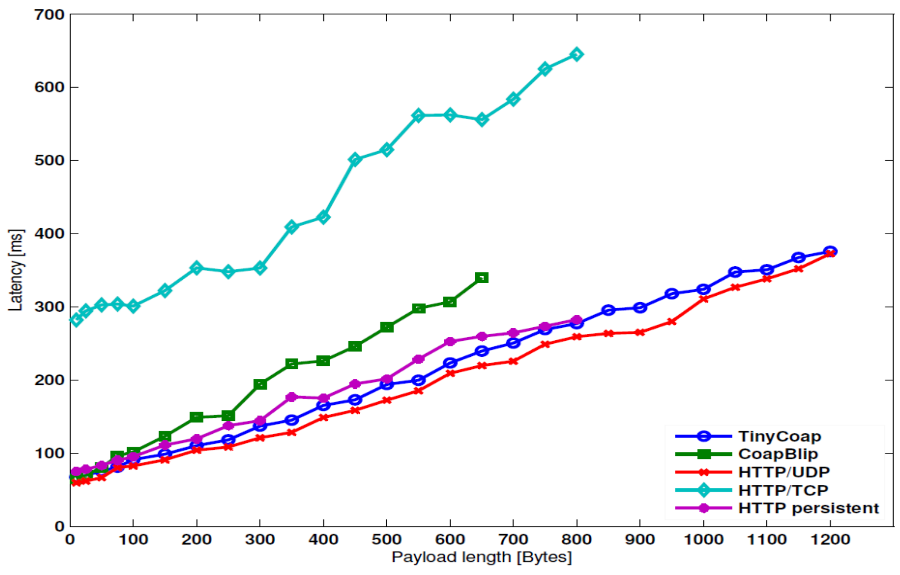
The HTTP/UDP implementation has the lowest memory footprint. This solution has a very low complexity and provides no reliability mechanism or request/response matching. The code size can therefore be minimized and ROM memory occupation can be reduced. Moreover, it is able to reach the minimal RAM memory occupation since it does not implement any HTTP buffer. This implementation uses exclusively the UDP buffer provided by Blip.

6.2. Latency

The latency experienced by a client on retrieving information from a server is one of the most important parameters used to evaluate the goodness of the server implementation. Low latency values can significantly enhance user experience and benefit those applications that work in real-time. We define latency as the time elapsed from the moment the client sends a request until the moment it receives the response.

Figure 6 shows the latency for each tested solution. Each point on the graph represents the average latency value of 100 successful request/response transactions. Payload size ranges from 10 to 1,200 bytes with increments of 50 bytes. The client sends a new request after receiving a response to the request previously sent. Figure 7 shows only a portion of the latency trend, excluding that obtained in HTTP/TCP. In this way, the differences between the other implementations can be better appreciated.

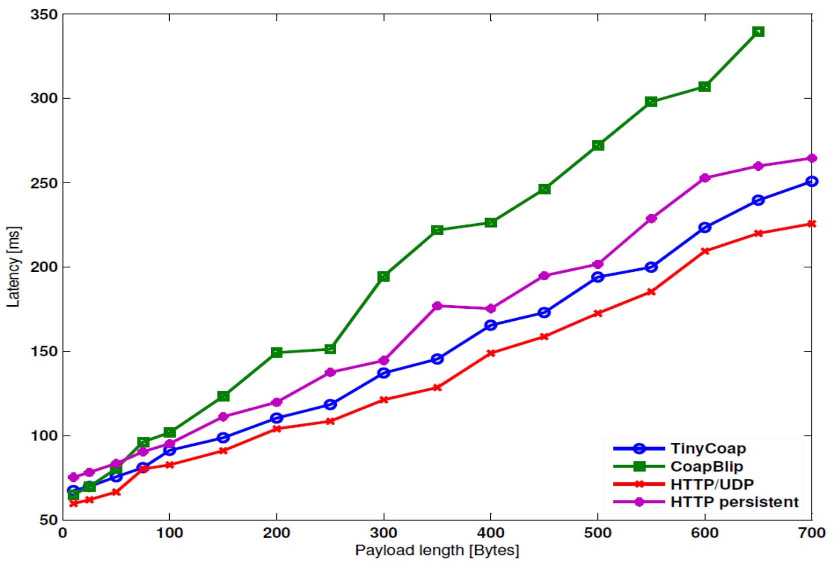
Figure 6. Latency evolution according to the payload size. The performance of HTTP increases significantly if a TCP persistent connection is used.



As expected, HTTP/TCP has the worst performance, the main reason for which is found in the latency introduced by the three-way handshake used by TCP to establish and close the connection.

However, the negative impact of the handshake can be reduced using the same TCP connection for various HTTP transactions. The performance obtained by HTTP persistent confirms this aspect. As may be seen in Figure 6, a server that uses HTTP persistent is able to lower the latency and to make its trend closer to those of the fastest solutions.

Figure 7. Details of the latency results. The better memory management of TinyCoAP allows the performance of CoapBlip to be improved.



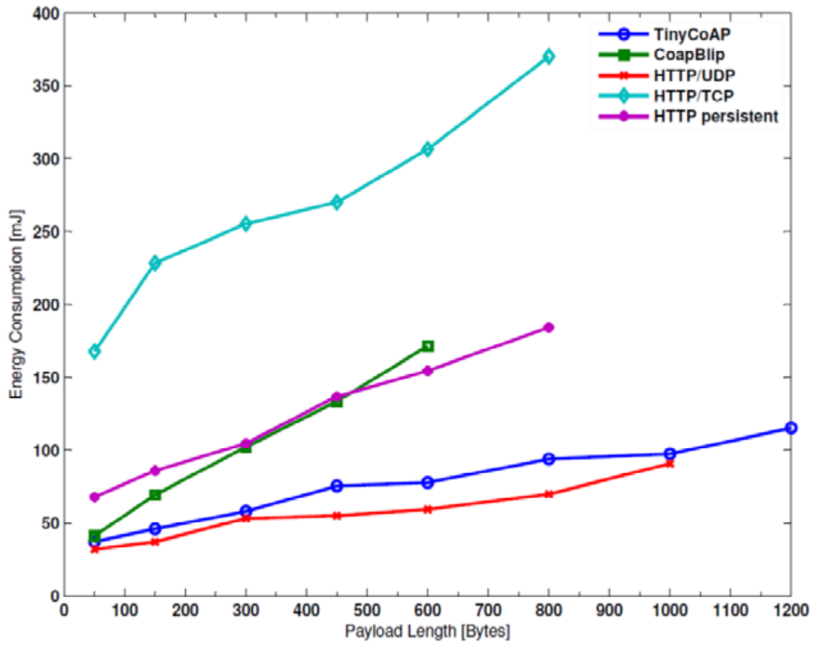
The lowest latency trend is obtained by the HTTP/UDP solution. This solution implements a bare HTTP server able only to reply to GET requests by querying for the provided resource. HTTP/UDP does not implement any reliability mechanism or HTTP logic. Therefore, it should be considered as a lower bound for latency.

TinyCoAP has a latency trend very close to that of HTTP/UDP, so the substantial growth of complexity of TinyCoAP is not reflected in a particular worsening of latency. TinyCoAP outperforms CoapBlip in terms of latency. The main reason for this is found in the enhanced management of RAM memory implemented in TinyCoAP. The memory allocation used in CoapBlip causes a growth of the packet processing time and limits the maximum payload size that CoapBlip is able to send to 650 bytes. Applications that use data aggregation or work with high payload sizes cannot be used either in CoapBlip or with HTTP solutions using TCP. Concerning HTTP, the maximum allowed payload size is limited by the use of the TCP buffers that exhaust the RAM memory.

6.3. Energy

Figure 8 shows the results obtained in the energy consumption tests. The results are obtained by measuring the energy consumed by a node when replying to ten consecutive requests. In fact, a fair comparison that may account for the persistent TCP connection used in HTTP persistent requires measurement of the energy consumed in more than one transaction.

Figure 8. Energy consumption. TinyCoAP has a lightweight packet processing that allows the energy consumption approaching the trend of HTTP/UDP to be lowered.



The results of this test do not take into account the energy wasted by the radio chip for listening the channel. Consequently, our evaluation does not need to consider power-saving protocols for radio duty-cycling. We only measured the energy consumed for receiving, processing and sending a packet. As mentioned in section IV the energy consumed by the CPU for packet processing is considerable. However, the energy consumed by the radio chip for receiving and sending a packet is still predominant over that used in packet processing. In this test, the consumption due to the radio chip has the same impact on the results of each implementation. The differences between the performance of each implementation are only due to the effects that the packet processing has on consumption. For each different payload size, we ran tests sampling the energy consumption each 0.02 ms. The device used for these measures is the Agilent Technologies DC power Analyzer N67705A.

As expected, a mote implementing HTTP solutions based on TCP consumes more energy than others. Once again, the message overhead caused by TCP proves to be costly for constrained networks. As seen in the latency tests, the HTTP persistent improves the performance of HTTP/TCP. This is due to the use of a persistent TCP connection. However, its performance is still much worse than that obtained by TinyCoAP or HTTP/UDP. The management of TCP connections requires a high degree of complexity and the maintenance in memory of the connection state. Consequently, there is a growth in the energy drawn by the RAM memory for keeping these states.

CoapBlip has a performance comparable to that of HTTP persistent. The onset of the CoapBlip trend is lower than that of HTTP persistent and comparable with that of TinyCoAP or HTTP/UDP. However, the increase in packet size causes a sharp rise in the CoapBlip energy trend, which exceeds that of HTTP persistent at 500 bytes. Once more, the mechanism implemented in CoapBlip for allocating and managing RAM memory proves to be unsuitable for constrained devices. Instead, TinyCoAP benefits from the different memory allocation mechanism in use. Furthermore, the different design of TinyCoAP allow to it to have less impact on energy consumption. Similarly to that seen in the latency tests, TinyCoAP has a performance that is highly comparable to that of HTTP/UDP. This proves once again that TinyCoAP is able to minimize the consumption of resources.

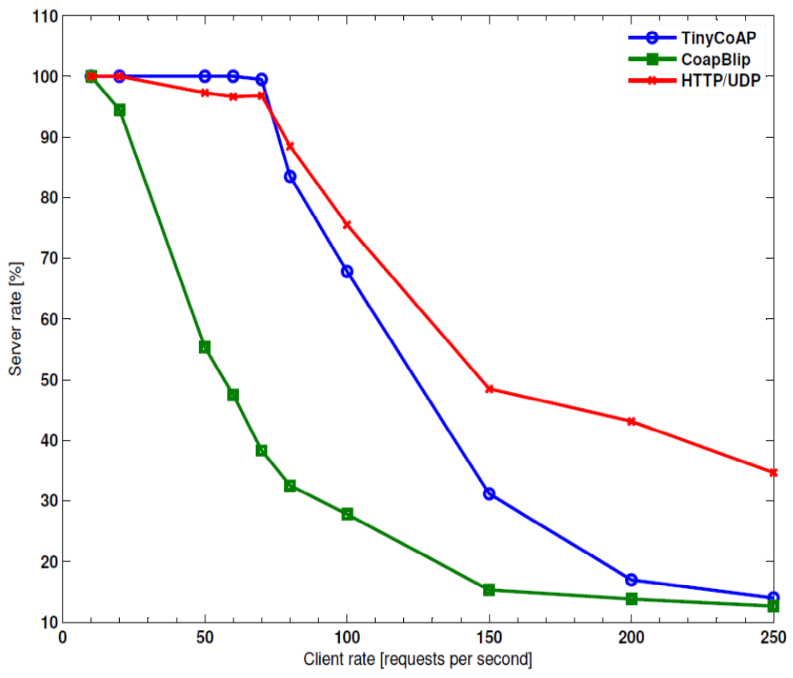
6.4. Workload

Comprehensive benchmarking requires a study of the effects that high workloads have on a server. In this study, we evaluate the percentage of requests that a server can handle as a function of the request rate of the client. We refer to the percentage of the handled requests with the term server rate. The test setup considers a client sending GET requests at a fixed rate. For each rate we run 100 tests. The resulting server rate is the average percentage of requests at which the server replies correctly in each test. The payload of the server response is greater enough to fill an IEEE 802.15.4 frame.

In this test, we evaluate only unreliable solutions. The presence of retransmissions does not allow maintaining constant the client rate. A client using CoAP reliability is forced to stop sending requests and start retransmitting. The server therefore would not be subjected to a constant workload. As a consequence, we use CoAP NON messages. The only reliability mechanism in use is that of the MAC layer.

We do not consider the use of HTTP/TCP. At present, the implementation of TCP does not allow accepting more than one connection at time. As a consequence, the rate with which a client sends requests is extremely limited. The average latency for HTTP/TCP considering a response sent in a full 802.15.4 frame is equal to 302 ms. Therefore, the maximum rate a client can have is estimated to 3 requests per second. We consider this rate too low for being compared with other solutions. Also the HTTP persistent has not been taken into account. In this solution, the rate at which a client sends requests cannot be controlled. It is the TCP congestion control algorithm that manages this rate. Figure 9 shows the average success rate of each considered solution.

Figure 9. Number of requests handled by the server as a function of the client rate. The enhanced buffers of TinyCoAP allows to that to overcome the CoapBlip performance.



The server rate performance is affected by the buffer size of the nodes and the mechanism used to access to the channel. A request or response could find the sending buffer full and, therefore, it would be discarded. This is more likely to happen when a client generates requests at a rate greater than that at which it can send them. Besides the processing time, the propagation time and the CSMA-CA mechanism are the main limitations of this rate. The propagation time account for the time needed to send the requests and to receive the MAC acknowledgment. A client has to wait for the server to acknowledge the receipt of the frame before sending another. In this work we use the unslotted CSMA-CA [2] of the IEEE 802.15.4 standard. This defines the procedure that the MAC and PHY layers have to follow for sending a packet. The node waits for a random number of backoff slots, then it perform the clear channel assessments (CCA). Should the client find the channel idle during the CCA period, it sends the request and then switches from the transmitting to receiving mode in a time that is called turnaround. Therefore, if the packet generation time were lower than that taken for the CSMA-CA plus the propagation time, then the client would fill its buffer faster than it can empty it. According to [48] this time is estimated to 6.09 ms. This is comparable to the packet generation time relative to a client rate of 150 requests per second. This time, however, is only an estimation. We expect that collisions are frequent in a congested wireless link. These imply MAC retransmissions and, consequently, a growth of the time needed for completing a request/response transaction. Packet collisions are also the main cause of the server rate degradation. These could happen when a node senses the channel idle during the turnaround time of its pair. Therefore, a request and a response could collide. This phenomenon is more likely to occur when the client rate is high. Furthermore, the channel becomes quickly congested and therefore the probability that a node senses it busy is high. The node give up the transmission of a packet if it senses the channel busy more than the maximum allowed.

The explanations given above are valid for all the tested solutions. The cause of the different performance has to be found in the packet processing and memory allocation of each one. Regarding HTTP/UDP, its light and fast processing allows to this solution to achieve the best performance. The server is able to process fast the received requests and therefore augments the server rate. The HTTP/UDP processing also helps improving the client performance. The packets stay for less time in the buffer and, therefore, a new generated packet has more possibility to find space respect to other solutions. This is also the cause for the less pronounced drop that HTTP/UDP has respect to the CoAP based solutions.

In Figure 9 one may observe that when the request rate become higher than 80, the success rate of TinyCoAP and HTTP/UDP undergo a pronounced drop. In fact, starting form this point the latency of a request/response transaction is comparable to the time taken by the client to generate and send a request. The probability of having a collision between a request and a response is therefore high. Starting from 150 requests per second, the drop is less sharp. The nodes are subjected to a workload close to its limits. The client rate can be augmented and the success rate will not lower as much as one may expect. As we explained above, these client rates saturate quickly the buffer. The sending process of many requests fails and therefore they never leave the node. The number of requests that are sent correctly became stable. As a consequence the server rate shows small variations.

CoapBlip has the worst performance. The inefficient use of the RAM memory and the complex packet processing are the main limitation of its server rate. Moreover, the higher latency experienced by CoapBlip implies lower the point at which this become comparable to the client rate. As a consequence, packet collisions affect the server rate earlier than in the other solutions.

6.5. Reliability

The reliability of IP based Web communications is traditionally provided by the TCP protocol. However, those application protocols that are bound to UDP should provide reliability by themselves. In fact, providing reliable communications is of paramount importance, especially when using wireless links that are known to be prone to packet loss. The IEEE 802.15.4 standard offers only hop-by-hop reliability by implementing packet retransmission at data-link layer. The lack of end-to-end reliability must therefore be compensated by implementing it at higher layers.

As anticipated, CoAP provides end-to-end reliability using CON messages through a simple stop-and-wait retransmission mechanism with exponential backoff. Should the CON message not be acknowledged, the CoAP node that started the communication will retransmit the message. In [11], the value for the initial timeout is fixed to 2 seconds, while the maximum number of retransmission is fixed to 4. Moreover, the CoAP definition allows this value to be changed according to the average Round Trip Time (RTT). Although not suggested in the standard, we recommend fixing a lower bound for the initial timeout. This timeout should not expire before the MAC layer reaches its maximum number of retransmissions. In [2], this number is fixed to 3, while the value of the timeout is not specified. TinyOS fixes this timeout to 512 symbols, which is equivalent to 8.192 ms.

In order to ensure a fair comparison, we should use the same values of initial timeout for all the solutions and avoid any randomness in the calculation of the subsequent values. We adopt the initial timeout value specified in the RFC 6298 [46]. This value is fixed to 1 second and is big enough to ensure that the CoAP first retransmission is sent after the last one in the MAC layer. The maximum number of retransmissions is fixed to 4, while the maximum value for the TCP retransmission timer is fixed to 16 seconds. This value corresponds to the maximum retransmission timeout reached in CoAP communications.

The evaluation of the mechanisms that each solution offers to provide reliability is performed by considering data transfers in an IEEE 802.15.4 link under a Rayleigh fading model. The basis for the calculations required by this model to determine the packet loss is reported in [47]. In this work, the authors studied the impact of the signal to noise ratio (SNR) on the PHY-level packet loss rate of an 802.15.4 link. As reported in [27], the packet loss rate P is given by:

$$P = 1 - (1 - S)^{2m} \tag{1}$$

where S is the symbol error rate and m the length in bytes of the packet. The corresponding packet length in symbol is 2 m. The symbol error rate is strictly related to the bit error rate B. The latter contains the relationship with the SNR:

$$B = \frac{1}{2} - \left(1 - \sqrt{\frac{SNR}{1 + SNR}}\right) \tag{2}$$

In this way, given an SNR, we can easily calculate the related packet error rate. However, this error rate does not correspond to that relative to the application layer, but rather to that of the PHY layer. The calculation of the error rate seen by the application layer requires one more step. The probability C of having an erroneous packet at application layer should take into account retransmissions done at the MAC layer, and is given by:

$$C = P^{(r+1)} \tag{3}$$

where r is the maximum number of retransmissions allowed for the MAC layer.

However, the formula in Equation (3) gives the application layer packet error rate only for non-fragmented packets. Should a packet be fragmented, this error rate changes according to the number of fragments. Thus, if f fragments compose a packet, the CoAP packet error rate is given by:

$$C_f = \sum_{j=1}^{f} C \times (1-P)^{j-1} \tag{4}$$

The maximum number of fragments that Blip allows is fixed to 12. It should be pointed out that all the formulas are valid only for single or fragmented packets that occupy the entire space available in 802.15.4 frames. As regards the presence of the PHY header, the frame can reach a maximum size of 133 bytes. According to Equation (4), the error rate for a non-fragmented packet is obtained for f equal to one.

The tests done for reliability cover four different SNR: 1 dB, 1.5 dB, 2 dB and 2.5 dB. In Table 4 we report the corresponding application layer packet error rates.

Table 4. Application layer packet error rate. These values refer to a single or fragmented packet that occupies the entire space of a 802.15.4 frame. The value for f = 1 refers to a single non-fragmented packet. A packet can consist of a maximum of 12 fragments.

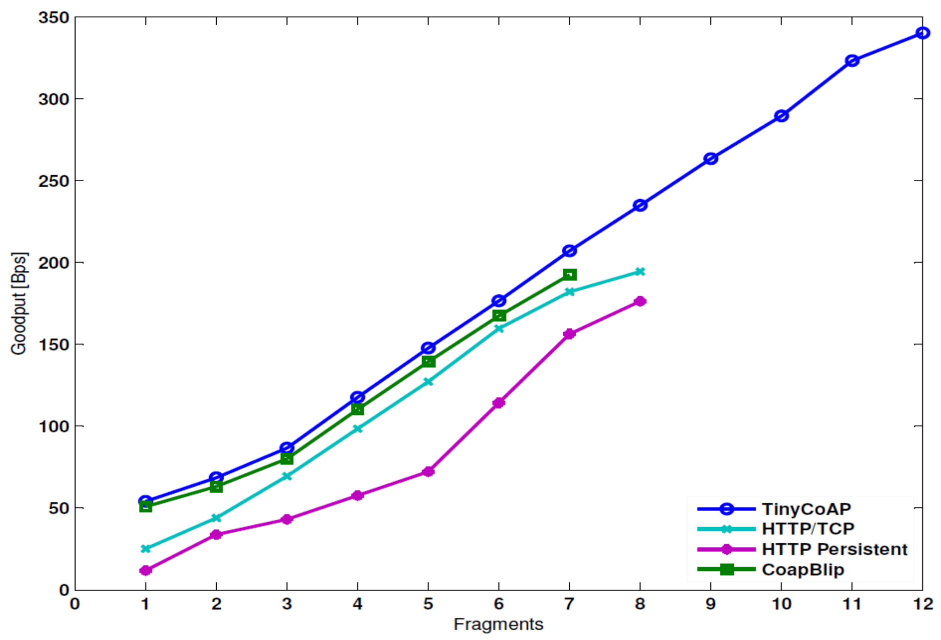
f	2.5 dB	2 dB	1.5 dB	1 dB
1	1.23%	5.20%	16.78%	39.64%
2	2.05%	7.91%	22.82%	47.82%
3	2.59%	9.33%	24.99%	49.51%
4	2.96%	10.08	25.77%	49.86%
5	3.20%	10.46%	26.06%	49.94%
6	3.37%	10.67%	26.16%	49.95%
7	3.47%	10.77%	26.19%	49.95%
8	3.55%	10.83%	26.21%	49.95%
9	3.59%	10.86%	26.21%	49.95%
10	3.63%	10.87%	26.21%	49.95%
11	3.65%	10.88%	26.21%	49.95%
12	3.66%	10.88%	26.21%	49.95%

According to the number of fragments and the relative error rate, we calculate the average goodput of a data transfer. This value is calculated according to the goodput obtained in 100 consecutive transactions. Unlike the previous tests, the reliability evaluation does not take into account the HTTP/UDP solution. In fact, none of those protocols provide any reliability mechanism, the definition of which is beyond the scope of this paper.

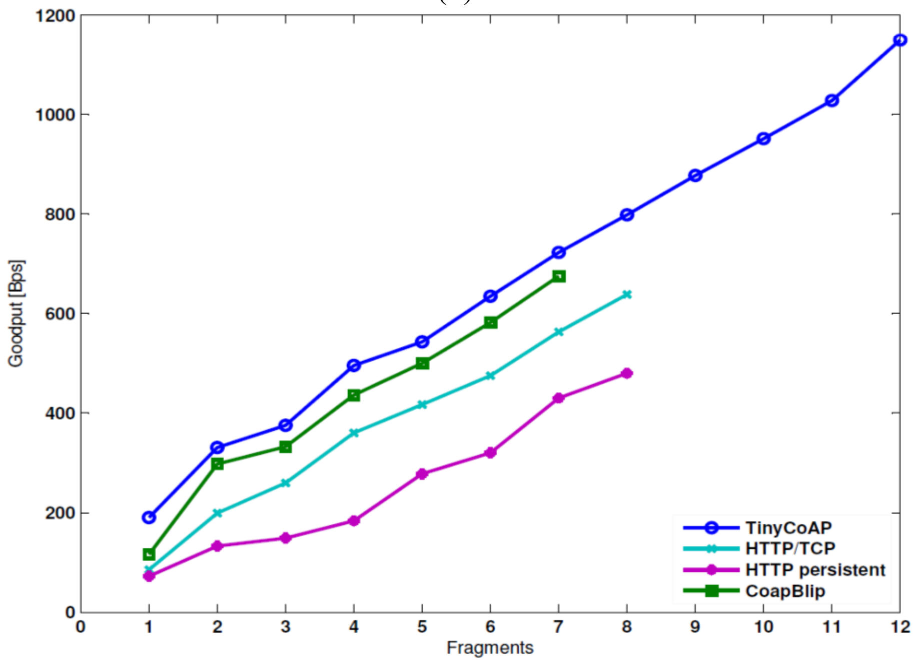
Errors are introduced by comparing the application layer error rates with a sequence of random numbers. Should the random number be less than or equal to the error rate, the CoAP or HTTP message received by the server is discarded at UDP layer. In this way, the application layer will consider that an error has occurred in the communication and will retransmit the packet. The same sequence of random numbers is used for each solution tested in order to ensure that in each experiment there are the same numbers of errors. Moreover, the errors will occur in the same order.

Figure 10 shows the goodput in bytes per second obtained for each SNR level. TinyCoAP surpasses the other solutions in all cases. CoapBlip has a goodput trend that is lower but close to that of TinyCoAP. In this case, the differences between the implementations of the retransmission mechanisms of TinyCoAP and CoapBlip are minimal and negligible. The different performances are therefore due exclusively to the differences in packet processing, as previously explained.

Figure 10. Goodput evolution in a channel under the Rayleigh fading model. (**a**) Goodput with SNR of 1 dB; (**b**) Goodput with SNR of 1.5 dB; (**c**) Goodput with SNR of 1.5 dB; (**d**) Goodput with SNR of 2.5 dB. The reliability mechanism implemented in CoAP yields a good performance. In TCP, the initial retransmission timeout is resettled after closing each connection, providing a better performance in channels with low SNR.

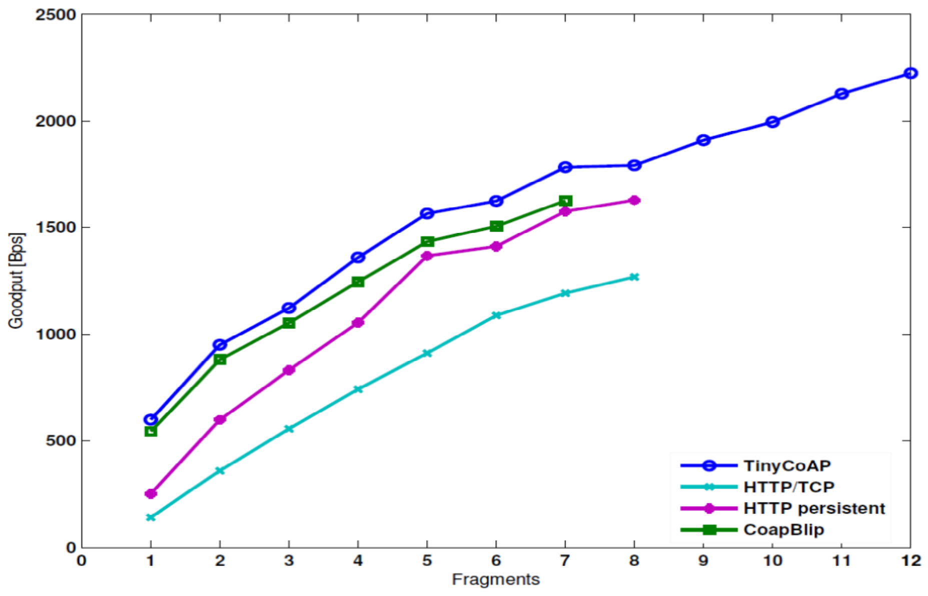


(**a**)

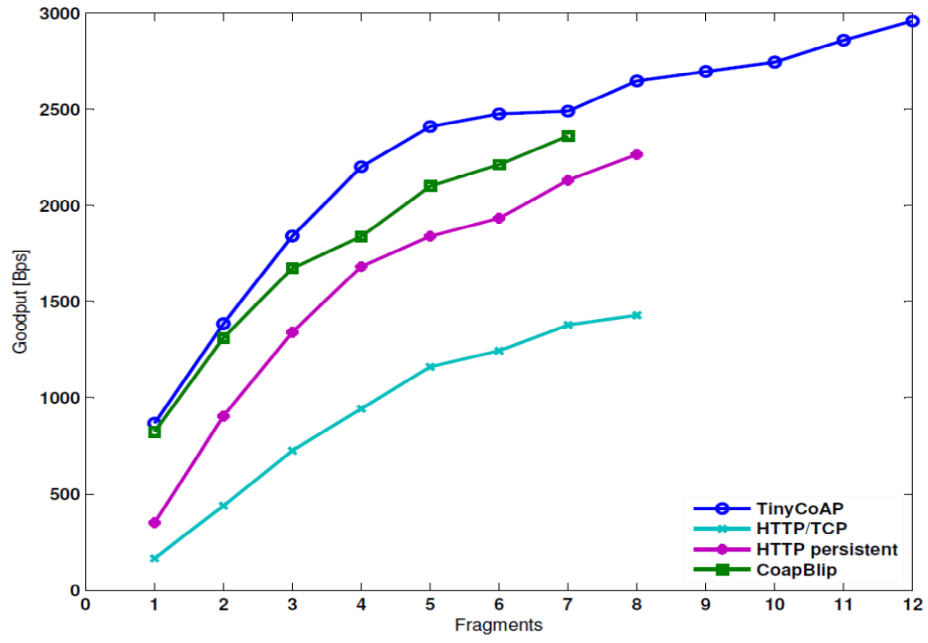


(**b**)

Figure 10. *Cont.*



(c)



(d)

Once again, the solutions for which HTTP is adopted yield the worst performance. In channels having a high SNR, the HTTP persistent has a goodput trend higher than that of HTTP/TCP. However, when the SNR is lower and the consequent error rate increases, the HTTP/TCP trend exceeds that of HTTP persistent. This can be explained by considering the behavior of the algorithm for the calculation of the retransmission timeout (RTO) specified in [46]. In fact, HTTP persistent and HTTP/TCP cannot measure the round trip time (RTT) of communications affected by retransmissions. The RTT is used by the algorithm to calculate the RTO and should be relative only to communications unaffected by retransmissions. However, the RTT of a retransmitted communication that has been acknowledged can only be measured by using the timestamp option of TCP. In our experiments, we employ a TCP version that does not use this option. Consequently, the behavior of HTTP/TCP and

HTTP persistent could differ significantly when the channel presents a low SNR and retransmissions are more frequent. Should two successive communications in HTTP persistent be affected by retransmissions, the RTO would not be recalculated but rather augmented exponentially until a valid RTT measure can be taken. Alternatively, HTTP/TCP would be able to measure the RTT and calculate the RTO each time it establishes a new connection. Thus, when there are successive communications affected by retransmissions, the HTTP persistent would undergo a latency greater than that of HTTP/TCP.

In conclusion, although CoAP is bound to the unreliable UDP protocol, the reliability mechanism it provides shows very good behavior and performance, at least in the channel affected by Rayleigh fading.

7. Conclusions

In this paper we present our original library for TinyOS, which we call TinyCoAP. We illustrate its design principles and describe its implementation. Furthermore, we compare it to the CoAP implementation distributed with TinyOS, called CoapBlip. The difference between the design of TinyCoAP and CoapBlip is also explained. Both solutions implement the recent defined CoAP protocol. This protocol aims at embedding RESTful Web services into WSNs. Besides CoAP, we also considered solutions based on the HTTP protocol.

All the solutions considered in this paper are discussed and evaluated in a real TinyOS based 6LoWPAN network. We measure the amount of memory occupied at compile time, the latency experienced by a client when retrieving information from a server, and the energy consumed when replying to the client. We also evaluated the performance of a server under high workloads. Finally, we evaluate the average goodput obtained in an 802.15.4 link under a Rayleigh fading model. In particular, the purpose of this test is to evaluate the reliability mechanism provided by CoAP. In our tests, we use HTTP with three different solutions for the transport layer. We consider the use of UDP, TCP and persistent TCP connections. We refer to each of these solutions as HTTP/TCP, HTTP persistent and HTTP/UDP.

HTTP/UDP is only able to reply to a simple HTTP message without implementing any logic behind it. The purpose of this implementation is to show a lower bound of the performance and to demonstrate how that of TinyCoAP and other solutions are close to it. HTTP/UDP has the best performance in terms of latency and energy. Although TinyCoAP is a more complex implementation, its performance is very similar to that of HTTP/UDP. This proves that its design is able to minimize the impact on the constrained resource of WSNs nodes while achieving good performance.

TinyCoAP provides the best performance in the rest of the considered parameters. In particular, TinyCoAP shows a significant improvement in performance compared with CoapBlip. The performance of CoapBlip is limited by the adoption of dynamic RAM memory allocation and the use of an external C library. Thanks to a design compliant to the TinyOS programming model and to the static allocation of memory, TinyCoAP solves the problems encountered in CoapBlip, thereby allowing CoAP to realize its full potential. TinyCoAP allows applications to work with a higher payload size than that achieved by CoapBlip or HTTP. This enables TinyCoAP to work with data aggregation, software update of nodes or video and audio applications that generate a high amount of data. Regarding HTTP, the performance obtained by HTTP persistent is an improvement on that provided by the traditional use of HTTP/TCP, which in any case is worse than that obtained by TinyCoAP.

In conclusion, TinyCoAP offers a lightweight, complete and flexible CoAP-based solution for implementing the Web communication paradigm in TinyOS based WSNs. TinyCoAP solves the problems experienced in CoapBlip, and is able to enhance performance significantly and to minimize the resource consumption. Part of our future work will focus on the design of a CoAP-HTTP proxy. In addition, we also plan to study mechanisms for managing CoAP client subscriptions to the resources offered by a server.

Acknowledgments

This work has been partially supported by the Spanish Government's Ministerio de Economia y Competitividad trough project TEC2012-32531, MICINN trough project TEC2009-11453, FEDER, the Catalan Government (Commissionat per a Universitat I Recerca del DIUE) and the European Social Budget.

Conflict of Interest

The authors declare no conflict of interest.

References

1. Hui, J.W.; Culler, D.E. Extending IP to low power, wireless personal area networks. *IEEE Internet Comput.* **2008**, *12*, 37–45.
2. IEEE. Wireless Medium Access Control (MAC) and Physical Layer (PHY) Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs). IEEE Standard 802.15.4–2006. Part 15.4; IEEE Computer Society: Los Alamitos, CA, USA, 2006.
3. RFC 6282. Compression Format for IPv6 Datagrams over IEEE 802.15.4-Based Networks. Available online. http://tools.ietf.org/html/rfc6282 (accessed on 11 March 2013).
4. Stirbu, V. Towards a RESTful Plug and Play Experience in the Web of Things. In Proceedings of IEEE International Conference on Semantic Computing, Santa Clara, CA, USA, 4–7 August 2008; pp. 512–517
5. Luckenbach, T.; Gober, P.; Arbanowski, S. TinyREST: A Protocol for Integrating Sensor Networks into the Internet. In Proceedings of the 1st Workshop on Real-World Wireless Sensor Networks (REALWSN), Stockolm, Sweden, 20–21 June 2005.
6. Fielding, R.T. Architectural Styles and the Design of Network-Based Software Architectures. Ph.D. Dissertation, University of California, Irvine, CA, USA, 2000.
7. Gudging, M.; Hadley, M.; Mendelsohn, N.; Moreau, J.; Nielsen, H.F.; Karmarkar, A.; Lafon, Y. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition). World Wide Web Consortium (W3C) Recommendation, April 2007.

8. Lerche, C.; Laum, N.; Moritz , G.; Zeeb, E.; Golatowski, F.; Timmermann, D. Implementing Powerful Web Services for Highly Resource-Constrained Devices. In Proceedings of 7th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), Seattle, WA, USA, 21–25 March 2011; pp. 332–335.
9. Groba, H.; Clarke, S. Web Services on Embedded Systems: A Performance Study. In Proceedings of 8th IEEE International Conference on Pervasive Computing and Communications Workshops (PERCOM Workshops), Mannheim, Germany, 29 March—2 April 2010; pp.726–731.
10. CoRE IETF Work Group. Available online: http://datatracker.ietf.org/wg/core/charter/ (accessed on 11 March 2013).
11. Shelby, Z.; Hartke, K.; Bormann, C.; Frank, B. Constrained Application Protocol (CoAP), draft-ietf-core-coap-13. Available online: https://datatracker.ietf.org/doc/draft-ietf-core-coap/ (accessed on 11 March 2013)
12. Kovatsch, M.; Mayer, S.; Ostermaier, B. Moving Application Logic from the Firmware to the Cloud: Towards the Thin Server Architecture for the Internet of Things. In Proceedings of the 6th International Conference on Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS 2012), Palermo, Italy, 4–6 July 2012
13. JCoAP. Available online: http://code.google.com/p/jcoap/ (accessed on 11 March 2013).
14. CoAPy. Available online: http://coapy.sourceforge.net/ (accessed on 11 March 2013).
15. Kuladinithi, K.; Bergmann, O.; Pötsch, T.; Beckera, M.; Görg, C. Implementation of CoAP and its Application in Transport Logistics. In Proceedings of Extending the Internet to Low power and Lossy Networks (IP+SN 2011), Chicago, IL, USA, 11 April 2011.
16. Kovatsch, M.; Duquennoy, S.; Dunkels, A. A Low-Power CoAP for Contiki. In Proceedings of Eighth IEEE International Conference on Mobile Ad-Hoc and Sensor Systems, Valencia, Spain, 17–22 October 2011; pp. 855–860.
17. Castellani, A.P.; Gheda, M.; Bui, N.; Rossi, M.; Zorzi, M. Web Services for the Internet of Things through CoAP and EXI. In Proceedings of IEEE International Conference on Communications Workshops (ICC), Kyoto, Japan, 5–9 June 2011; pp 1–6.
18. Silva, R.; Sá Silva, J.; Boavida, F. Evaluating 6LoWPAN Implementations in WSNs. In Proceedings of the 9th Conference on Computer Networks, Oeiras, Portugal, 15–16 October 2009.
19. Yibo, C.; Hou, K.; Zhou, H.; Shi, H.; Liu, X.; Diao, X.; Ding, H.; Li, J.; De Vaulx, C. 6LoWPAN Stacks: A Survey. In Proceedings of 7th International Conference on Wireless Communications, Networking and Mobile Computing (WiCOM), 2011, Wuhan, China, 23–25 September 2011; pp.1–4.
20. Colitti, W.; Steenhaut, K.; De Caro, N. Integrating Wireless Sensor Networks with the Web. In Proceedings of Extending the Internet to Low power and Lossy Networks (IP+SN 2011), Chicago, IL, USA, 11 April 2011.
21. Duquennoy, S.; Wirstom, N.; Tsiftes, N.; Dunkels, A. Leveraging IP for Sensor Network Deployment. In Proceedings of Extending the Internet to Low power and Lossy Networks (IP+SN 2011), Chicago, IL, USA, 11 April 2011.

22. Colitti, W.; Steenhaut, K.; De Caro, N.; Buta, B.; Dobrota, V. Evaluation of Constrained Application Protocol for Wireless Sensor Networks. In Proceedings of 18th IEEE International Workshop of Local and Metropolitan Area Networks (LanMan), Chapel Hill, NC, USA, 13–14 October 2011.
23. Duquennoy, S.; Grimaud, G.; Vandewalle, J.-J. Smews: Smart and Mobile Embedded Web Server. In Proceedings of International Conference on Complex, Intelligent and Software Intensive Systems, CISIS'09, Fukuoka, Japan, 16–19 March 2009; pp. 571–576.
24. Sugoog, S. Protocol implementations for web based control systems. *Int. J. Control Autom. Syst.* **2005**, *3*, 122–129.
25. Dunkels, A. Full TCP/IP for 8-Bit Architectures. In Proceedings of the 1st International Conference on Mobile Systems, Applications and Services MobiSys'03, San Francisco, CA, USA, 5–8 May 2003; pp. 85–98
26. Lin, T.; Zhao, H; Wang, J.; Han, G.; Wang, J. An Embedded Web Server for Equipment. In Proceedings of 7th International Symposium on Parallel Architectures, Algorithms and Networks, Hong Kong, China, 10–12 May 2004; pp. 345–350.
27. Agranat, I.D. Engineering Web technologies for embedded applications. *IEEE Int. Comput.* **1998**, *2*, 40–45.
28. Klimchynski, I. Extensible embedded web server architecture for Internet-based data acquisition and control. *IEEE Sens. J.* **2006**, *6*, 804–811.
29. Priyantha, N.B.; Kansal, A.; Goraczko, M.; Zhao, F. Tiny Web Services: Design and Implementation of Interoperable and Evolvable Sensor Networks. In Proceedings of the 6th ACM Conference on Embedded Network Sensor Systems, SenSys'08, Rayleigh, NC, USA, 4–7 November 2008; pp. 253–266.
30. Shelby, Z. Embedded web services. *IEEE Wirel. Commun.* **2010**, *17*, 52–57.
31. Tolle, G. Embedded Binary HTTP (EBHTTP). Available online: http://tools.ietf.org/id/draft-tolle-core-ebhttp-00.txt (accessed on 11 March 2013).
32. Dawson-Haggerty, S.; Jiang, X.; Tolle, G.; Ortiz, J.; Culler, D. sMAP—a Simple Measurement and Actuation Profile for Physical Information. In Proceedings of the 8th ACM Conference on Embedded Networked Sensor Systems (SenSys'10), Zurich, Switzerland, 3–5 November 2010.
33. Frank, B. Chopan: Compressed HTTP over PANs. Available online: http://tools.ietf.org/html/draft-frank-6lowapp-chopan-00 (accessed on 11 March 2013).
34. Ludovici, A.; Calveras, A. Integration of Wireless Sensor Networks in IP-based networks through Web Services. In Proceedings of 4th Symposium of Ubiquitous Computing and Ambient Intelligence, Valencia, Spain, 7–10 September 2010.
35. Chander, R.P.V.; Elias, S.; Shivashankar, S.; Manoj, P. A REST based Design for Web of Things in Smart Environments. In Proceedings of 2nd IEEE International Conference on Parallel Distributed and Grid Computing (PDGC), Waknaghat, Solan, Himachal Pradesh, India, 6–8 December 2012; pp. 337–342.
36. Lerche, C.; Hartke, K.; Kovatsch, M. Industry Adoption of the Internet of Things: A Constrained Application Protocol Survey. In Proceedings of the 7th International Workshop on Service Oriented Architectures in Converging Networked Environments (SOCNE 2012), Kraków, Poland, 17–21 September 2012.

37. Harvan, M.; Schoenwaelder, J. TinyOS Motes on the Internet: IPv6 over 802.15.4 (6lowpan). *Praxis der Informationsverarbeitung und Kommunikation (PIK)* **2008**, *31*, 244–251.
38. Potsch, T.; Kuladinithi, K.; Becker, M.; Trenkamp, P.; Goerg, C. Performance Evaluation of CoAP Using RPL and LPL in TinyOS. In Proceedings of 5th International Conference on New Technologies, Mobility and Security (NTMS), Istanbul, Turkey, 7–10 May 2012
39. Levis, P.; Madden, S.; Polastre, J.; Szewczyk, R.; Woo, A.; Gay, D.; Hill, J.; Welsh, M.; Brewer, E.; Culler, D. TinyOS: An Operating System for Sensor Networks. In *Ambient Intelligence*; Springer: Berlin, Germany, 2005; pp. 115–148.
40. Gay, D.; Levis, P.; Von Behren, R.; Welsh, M.; Brewer, E.; Culler, D. The nesC language: A holistic approach to networked embedded systems. *ACM SIGPLAN Not.* **2003**, *38*, 1–11.
41. Shelby, Z. CoRE Link Format. Available online: http://tools.ietf.org/html/draft-ietf-core-link-format-13 (accessed on 11 March 2013).
42. Hartke, K. Observing Resources in CoAP. Available online: http://tools.ietf.org/html/draft-ietf-core-observe-05 (accessed on 11 March 2013).
43. Levis, P. TinyOS Programming. Available online: http://csl.stanford.edu/~pal/pubs/tinyos-programming.pdf (accessed on 11 March 2013).
44. Blip. Available online: http://smote.cs.berkeley.edu:8000/tracenv/wiki/blip (accessed on 11 March 2013).
45. Crossbow Technology Inc., TelosB Datasheet. Available online: http://www.willow.co.uk/TelosB_Datasheet.pdf (accessed on 11 March 2013).
46. RFC6298, Computing TCP’s Retransmission Timer. Available online: http://tools.ietf.org/html/rfc6298 (accessed on 11 March 2013).
47. Goyal, M.; Prakash, S.; Xie, W.; Bashir, Y.; Hosseini, H.; Durresi, A. Evaluating the Impact of Signal to Noise Ratio on IEEE 802.15.4 PHY-Level Packet Loss Rate. In Proceedings of 13th International Conference on Network-Based Information Systems, Takayama, Japan, 14–16 September 2010.
48. Lauwens, B.; Scheers, B.; Van de Capelle, A. Performance analysis of Unslotted CSMA/CA in wireless networks. *Telecommun. Syst.* **2010**, *44*, 109–123.