Alma Mater Studiorum - Università di Bologna

FACOLTÀ DI SCIENZE MATEMATICHE, FISICHE E NATURALI

Corso di Laurea Magistrale in Informatica

Design e implementazione di un proxy HTTP/CoAP

Presentata da:

Mirko Rossini

Relatore:

Chiar. mo Prof. **Renzo Davoli**

Correlatore:

Dott. **Thomas Fossati**

Sessione III
Anno Accademico 2010-2011

Design e implementazione di un proxy HTTP/CoAP

Mirko Rossini

February 26, 2012

Contents

List of Figures

Capitolo 1

Introduzione

Nell'ultimo decennio, soprattutto grazie alla nascita di smartphone e tablet, il numero di apparecchi fisici connessi alla rete è enormemente aumentato. Questa tendenza ha fatto sì che, nel 2008, il numero di dispositivi dotati di funzioni di connettività alla rete superasse la popolazione terrestre. Il momento storico di questo superamento, per convenzione, si usa come punto di riferimento per indicare la data di nascita di Internet of Things.

Il termine internet of Things si riferisce all'estensione di internet al mondo delle cose, trasformando gli oggetti, in particolare sensori e attuatori, in una risorsa informativa integrata nella piattaforma Web.

Secondo le proiezioni di Cisco, Internet of Things sarà composta da 25 miliardi di dispositivi interconnessi nel 2015, e da 50 miliardi nel 2020 [13]. Queste quantità sono giustificate dagli indubbi vantaggi che le reti di oggetti porterebbero all'umanità: molte delle applicazioni di IoT richiedono delle reti di grandi dimensioni (con un considerevole numero di nodi interconnessi con collegamenti wireless), costituite da piccoli apparecchi a basso costo e consumi ridotti.

Le applicazioni delle reti basate su Internet of Things cambieranno radicalmente l'umanità. L'esempio più citato è quello della domotica, dove sensori e attuatori, collegati in reti wireless, potranno essere utilizzati per il miglioramento della qualità della vita (monitorando costantemente l'ambiente casalingo e le sue condizioni) e per il risparmio energetico. Sempre per il risparmio energetico, si potranno utilizzare i dispositivi connessi per realizzare le cosiddette "Smart Grid". In una Smart Grid, gli elettrodomestici, e in generale gli apparecchi elettrici, possono comunicare con delle unità di gestione, installate ad esempio in una centrale elettrica. Lo scopo di questo scambio di informazioni è fare si che gli elettrodomestici possano distribuire il carico di richiesta elettrica nell'arco della giornata, così da evitare picchi di consumo.

Un altro esempio di applicazione di IoT è la medicina. Ogni persona potrà utilizzare sensori e apparecchi di vario genere (bilance, monitor cardiaci) per essere costantemente seguita durante una cura o per effettuare una diagnosi precoce.

Reti di sensori possono anche essere utilizzate per applicationi su vasta scala come il controllo ambientale, i trasporti e la logistica. Ad esempio, si potranno installare reti di sensori anti-incendio distribuiti su ampie aree forestali, o rilevatori di traffico per gestire l'invio di merci in maniera efficiente.

Affinchè Internet of Things abbia uno sviluppo che ricalchi quello ipotizzato, si dovranno superare una serie di difficoltà. La prima e più complessa è rappresentata dall'energia. Se verranno rispettate le proiezioni, il numero di apparecchi connessi alla rete sarà tale da renderne proibitiva la gestione manuale dell'alimentazione. In altre parole, cambiare le batterie a miliardi di dispositivi richiederebbe un lavoro, ed un relativo costo, tali da oscurare i vantaggi reali derivanti dalle reti di oggetti. Saranno quindi necessarie delle tecnologie che rendano gli apparecchi autosufficienti, ad esempio convertendo luce, vento e vibrazioni in energia.

Queste nuove reti di piccoli dispositivi, per i motivi indicati precedentemente, dovranno avere un ridotto consumo di energia. Per questo motivo, le capacità computazionali, e la potenza dei dispositivi di rete, saranno estremamente limitate, andando a creare un ambiente definito "constrained". Questo tipo di ambiente richiede la stesura di standard che prestino grande attenzione all'uso delle risorse disponibili. Per lo sviluppo di Internet of Things, è necessario che vengano ultimati, in questi anni, tutti gli standard necessari.

Come traspare da quanto visto finora, Internet of Things e Internet sono due entità distinte, basate su standard differenti e con diverse caratteristiche fisiche. Per i futuri sviluppi della rete, sarà però indispensabile progettare e implementare sistemi che siano in grado di integrare IoT e Internet. Questa integrazione consiste nel permettere a una o più reti wireless di sensori ed attuatori di interagire con la rete Internet in maniera semplice e trasparente. L'interoperabilità di IoT e Internet, permetterà di realizzare semplicemente interfacce di interazione, uomo-macchina e macchina-macchina, riutilizzando la preesistente struttura della Rete e la sua componentistica. Questo approccio permette di convergere i dati provenienti o diretti alle "cose" all'interno di un layer applicativo già molto ricco, grazie allo sviluppo seguito dal web negli anni passati.

Quest'ultimo aspetto è fulcro di questo documento. Nei successivi capitoli, si esporrà una analisi sullo stato dell'arte dell'Internet degli oggetti. Dopo aver parlato più nel dettaglio di cos'è IoT e di come è realizzabile,

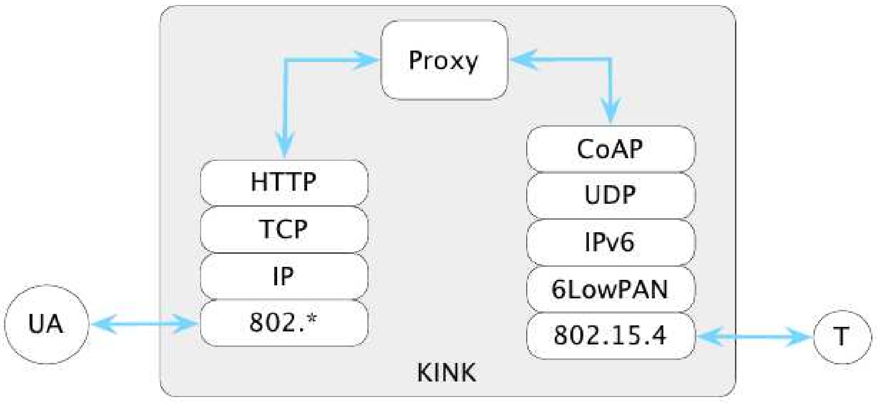


Figura 1.1: Kink come proxy per integrare IoT e Internet

verranno descritti i principali protocolli utilizzati all'interno delle reti wireless di sensori/attuatori. Verranno poi elencate alcune implementazioni esistenti di tali reti e le caratteristiche dei prodotti commerciali e proprietari che vengono usati attualmente per realizzarle. Ci si concentrerà poi sui protocolli aperti, in fase di definizione, che verranno utilizzati per gestire le future reti IoT: 6LoWPAN, che definisce le metodologie per trasportare pacchetti IPv6 in ambiente constrained, e CoAP, protocollo che descrive un insieme di funzionalità utili alla creazione di applicazioni su reti di oggetti.

Il progetto finale, che prende il nome di Kink, sarà in grado di integrare tutti gli strati delle due pile (di IoT ed Internet), a partire dalla traduzione di richieste HTTP in CoAP e vice versa. Dopo la presentazione dello stato dell'arte di IoT, si descriveranno nel dettaglio le fasi della progettazione e sviluppo di Kink. In particolare, verranno indicati i requisiti ed i vincoli del progetto, specialmente quelli fisici derivanti dall'ambiente constrained. Si discuterà poi il design di alto livello dell'applicazione, partendo dalla suddivisione delle aree funzionali del proxy in moduli software. Si descriveranno infine i dettagli di progetto dei componenti individuati, e le streategie di comunicazione tra gli stessi.

Attenzione relativamente ampia sarà dedicata ai dettagli progettistici ed implementativi del sistema di caching di Kink, fondamentale per adattare il prodotto finale all'hardware di esecuzione previsto (a basse capacità di memoria, disco e potenza computazionale).

Nelle ultime sezioni di questo docomento, verrà descritta la realizzazione di una applicazione dimostrativa che faccia uso di Kink, mettendo in comunicazione un normale browser con una piccola rete creata basandosi sugli standard di IoT descritti. I risultati di questa applicazione verranno esposti e discussi.

In ultimo, si esporranno le prospettive future di Kink, descrivendo inoltre dei miglioramenti, apportabili allo standard CoAP in sviluppo, emersi durante lo svolgimento del progetto.

Chapter 2

The Internet of Things

The Internet is one of the most important creations in history, as it revolutionized the human life being an immense source of free, and easily accessible, knowledge. While the Web changed in time, from being a military network research project to the social experience it is today, the Internet, in its history, haven't changed much, aside the regular improvements to the technology.

The Internet of Things, the "extension" of the Internet that allows objects to communicate with each other without the need of human interaction, is the first real evolution of the Internet. According to the Cisco Internet Business Solutions Group (IBSG), "IoT is simply the point in time when more "things or objects" were connected to the Internet than people"[13]. Connected devices outnumbered people between 2008 and 2009, giving birth to the Internet of Things.

2.1 Applications

The Internet of Things, and net devices network that it allows to create, has an enormous potential to revolutionize human life like the Internet first did. The applications that can be foreseen now are many and have a huge impact on everyday life, energy and the whole earth as well.

Home automation

Connected devices can be installed in every house. These devices can serve for many purposes:

- Remotely control the house air conditioning and heating, to smartly adjust the temperature exactly when needed, in order to save energy

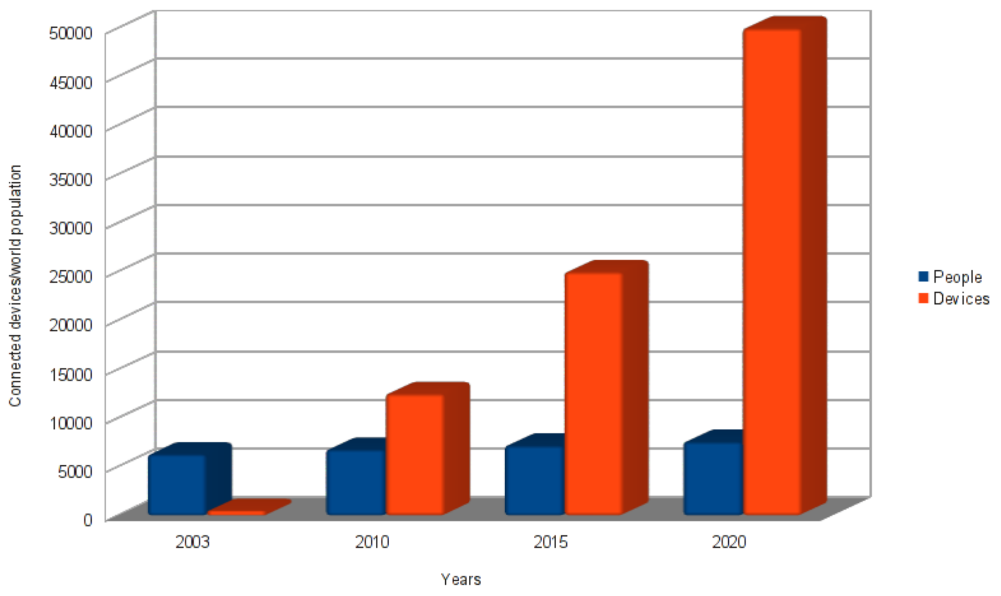


Figure 2.1: Number of devices connected to the Internet and earth population

- Control the heating of the water in the house to fit the occupant's needs
- Remotely monitor the quality of the air in the house
- Install intelligent actuators that behaves differently depending on external conditions (e.g. an alarm clock that wakes up its owner earlier in case of slow traffic)
- Get informations and reminders about the house, for example using a fridge that warns its owner when the expiration date of a stored product is approaching.

Health

Connected devices can be used for remote monitoring of patient conditions and early diagnosis. Examples of such devices are:

- Weighting machines
- Blood analyzers

- Cardiac monitors
- Blood pressure meters

Other devices can be used to monitor how the patient is following a treatment or a diet, and as a medication reminder.

Energy

The IoT can be used to leverage the usage of electricity, in order to avoid consumption peaks that create the need of unnecessary power implants. Devices, communicating with a central system, will be able to retrieve data about the cost of electricity, and begin to work when energy is less expensive.

Sensor nets

Sensors can be organized in networks to gather data over vast areas. Some examples of applications of sensor nets are:

- Early fire alarm in big forests
- Microphones, water analyzers, pressure meters placed in open sea to monitor the behavior of animals and water pollution
- Analysis of data for scientific research, interchanging data between research centers around the globe
- Traffic monitoring and regulation

2.2 Barriers to IoT

Several barriers may slow down the development of the IoT. A first issue is that all the sensors and devices will need an IP address to operate on the network. As the number of these new network nodes is going to be huge, using IPv6 is mandatory. IPv6 offers configuration capabilities and improved security features, on top of an addressing system capable of assigning $3,4 * 10^{38}$ unique addresses. The development of the IoT depends directly on the worldwide deployment of IPv6.

All the devices will need energy to operate, and as handling batteries in billions of devices is prohibitive, sensors will need to be self-sustaining. What's needed is a technology that allows sensors to generate electricity using the surrounding environment, for example they should gather energy from light, vibrations or the wind. Without an economically sustainable technology that can serve this purpose, the IoT will not reach its full potential.

The last barrier are standards. Much progress in the area of standards has been made (as will be shown in the following chapters), but some of these standards aren't mature enough yet, and more work is needed. The IoT will not be complete until these standards are fully defined.

In this document, we will present the state of the art of the IoT, and we will discuss the design and implementation of a "brick" of the IoT structure: a bridge between the Big Internet we are using today and the new Internet of small, cheap and constrained devices.

Chapter 3

Characteristics of the IoT WPANs

3.1 LoWPAN

While working with network devices in the IoT, we have to take into account that we are operating on a ”constrained” environment. This means that, the machines that will run the produced software, will not be as performant as those that operate on unconstrained environments such as the Big Internet networks. The IoT networks on which we intend to operate are called LowPANs: Low power WPANs (Wireless Personal Area Networks) . LoWPANs have typically the following characteristics:

- Exchanged packets are small in size
- Low bandwidth
- The network runs on low power, typically battery operated, and low cost devices
- LoWPANs must connect devices that are typically deployed in large numbers and in unpredictable locations
- The devices the LoWPAN consist of tend to be unreliable due to:
  - Uncertain radio connectivity
  - Battery drain
  - Device lockups
  - Physical tampering

- Network devices may sleep for long periods of time to save energy
- The networks are organized in topologies like:
  - Star
  - Mesh.

In the past years, many standards and software have been created to fit these kind of constraints environments.

3.2 The IEEE 802.15.4 standard

The IEEE 802.15.4 standard was created to define how the physical and MAC (Media Access Control) level of the ISO/OSI stack should be implemented to allow the creation on networks on embedded devices, such as the ones in LoWPANs, described in section 3.1. This standard was created and is currently maintained and updated by the IEE 802.15 group.

The general characteristics of the networks and devices the IEEE 802.15.4 was created for are:

- Data transfer rates ranging from 20 kbps to 250 kbps
- Communication range up to 10 meters
- Low power, cost and bandwidth

In the IEEE 802.15.4 standard, two different types of devices are considered:

- Reduced Function Device (RFD). RFDs are devices with reduced communicative and computational capacities, and can only talk to coordinators
- Full Function device (FFD). FFDs are devices that are relatively performant. They can act as coordinators of the network and can communicate with any other device regardless of its role.

Furthermore, the IEEE 802.15.4 standard describes two different network configurations: star or peer to peer. In the star topology, a FFD acts a coordinator for the WPAN, and is responsible for handling all communication between components. In the peer to peer topology, a device in the WPAN can communicate with any other device within its range. This usually entails that most of the of devices in a peer to peer 802.15.4 PAN are FFDs.

Data transfer

Data transfer, in the 802.15.4 standard, can be performed in two ways: from a coordinator to a network device and vice versa. Data transfer from network devices to coordinators is fairly simple and consists with these steps:

- The network device sends data to the coordinator
- The coordinator sends back an acknowledgment if the transfer was successfull.

Data transfer in the opposite way is slightly more complicated:

- The network device sends a data request
- The coordinator responds with an acknowledgment
- The coordinator sends the requested data
- The network device responds with an acknowledgment.

802.15.4 Layers

As mentioned before, the architecture of the IEEE 802.15.4 standard consists of two layers: the physical layer and the MAC layer. the OSI model.

The phisical layer

On the physical layer in the 802.15.4, are defined the features to allow WPAN devices to:

- Manage the radio transmitter
- Transmit data
- Receive data

The 802.15.4 standard physical layer operates over the following frequency bands:

- 868 to 868.8 MHz in Europe
- 902 to 928 MHz in the United States
- 2400 to 2483.5 MHz for the whole world.

Packets exchanged with the 802.15.4 physical layer consists of a header part, where are indicated the informations about synchronization and content length, and a payload. The header has a maximum size of 6 bytes and consists of three parts:

- Preamble (32 bits), that is used to synchronize the sender and receiver
- Start of Frame Delimiter (8 bits), that denotes the end of the preamble
- Length (8 bits), that indicates the length of the payload, with a maximum of 127 bytes.

The MAC layer

With the MAC layer, devices are able to perform all those operations needed to participate to a WPAN. In particular, the Medium Access Control layer handles:

- Association and dissociation to a WPAN
- Acknowledged frame delivery
- Access to channels
- Frame validation
- Beacon management
- Time slot management.

To handle these functinos, the MAC layer has four different frame types: the beacon frame, data frame, acknowledgment frame, and MAC command frame. Depending on the frame type, a MAC PDU may have a different format. The MAC PDU is composed of:

- The MAC header (MHR), wich is subdivided into:
  - Frame control (2 bytes), it specifies the format of the MAC frame being transmitted, indicating how the address field should be handled
  - Sequence number (1 byte), it is used to associate acknowledgement with the previously sent data packet

  - Addressing field (1-20 bytes), its format depends of the type of MAC packet being sent, for example, a request packet would indicate in the address field both the sender and the receiver, while a beacon message could only contain the sender's address

- The MAC service data unit (MSDU), that contain the actual data of the PDU. The data maximum length is 116 bytes

- The MAC footer (MFR), that contains the FCS (Frame Check Sequence), a 2 bytes string used to validate incoming data packets.

Chapter 4

Non-IP and Proprietary solutions

The IEEE 802.15.4 standard is implemented by some proprietary solutions, presented in the next sections. These solutions also offer an implementation of the upper layers of the ISO/OSI model. These projects provide a complete solution to create WPANs using embedded devices.

4.1 WirelessHART

WirelessHART is a proprietary wireless device communication technology based on the Highway Addressable Remote Transducer Protocol (HART). WirelessHART is based on the IEEE 802.15.4 standard, and was developed, starting from early 2004, by 37 companies forming the HART Communications Foundation[7] (HCF).

HART provides a bi-directional communication system that allows intelligent instruments and host systems to exchange data. The WirelessHART protocol is developed to allow communication among devices with a master/slave policy: slave devices only speak when requested by a master.

To get access to protocol specifications, an company must first join the HART Communication Foundation, paying a fee starting from 6000$.

4.2 Zigbee

ZigBee[10] is the name of a specification, for a set of communication protocols, for low-power digital radio devices, based on the IEEE 802.15.4 standard. The specification is freely available for non-commercial purposes.

In order to create products for market with ZigBee, a developer must first become a member (paying a fee) of a group of companies, the ZigBee alliance, that maintain and publish the ZigBee standard. The specifications released as of today are:

- ZigBee Home Automation
- ZigBee Smart Energy 1.0
- ZigBee Telecommunication Services
- ZigBee Health Care
- ZigBee RF4CE - Remote Control
- ZigBee Smart Energy 2.0 (under development)
- ZigBee Building Automation (under development)
- ZigBee Retail Services (under development)

4.3 MiWi

MiWi[2] and MiWi P2P are proprietary wireless protocols based on the IEEE 802.15.4 standard. MiWi was designed by Microchip Technology as a network solution to work on small, low-power digital radio devices.

MiWi has a very small footprint, and is designed to be used on the PIC platform On the PIC platform, a micro controller family created by Microchip Technology itself.

The Miwi software can be freely downloaded, but developers are obliged to use it only with Microchip microcontrollers.

4.4 Ad-hoc solutions

Besides the proprietary solutions described in chapter 4, some ad-hoc solution have been developed to address the problem of retrieving data from intelligent (sensor) networks, and provide human usable interfaces on those data. In the following sections, a few examples of these ad-hoc solutions will be presented.

4.5 FARO (Fast Access to Remote Object)

FARO is a software system, developed by the ENEA agency, that centralizes the access to structures, resources and services through the web[21]. The components of the FARO structure are:

- NX technology, a proprietary software, developed by NoMachine, that can be used to perform secure remote access and desktop virtualization
- A GUI, written in Java, that can be customized according to the user needs
- Modules that implement the actual communication to the resources.

The FARO technology has been implemented, and is currently in production, to provide interfaces on many structures. The most significant examples are:

- FARO is used by ENEA to provide access to its grid computing infrastructure
- A version of FARO is used by the EFDA (European Fusion Development Agreement) researchers for many tasks, like analysis and simulation
- Another version is used for ARK3D (The ENEA-GRID infrastructure for the Remote 3D)

4.6 THREDDS (THematic Real-time Environmental Distributed Data Services)

THREDDS is a middleware created to generally fill the gap between data providers (generically, systems connected to sensor nets) and data users (humans or machines that handle the data in some way)[19]. THREDDS allows data users to find the data sets they need, retrieve and use them. Data providers can publish a catalog of the available data. Data sets are provided in XML format.

Catalogs can be generated locally or remotely and the data generation can be executed automatically at times decided by the data providers. The final purpose of the project is to install a IOOS (Integrated Ocean Observing

System), taking as example the analogous project coordinated by NOAA in the USA.

Chapter 5

6LoWPAN and REST solutions

As we want our product to be free and distributed under an open source license, we must make use of compatible instruments. Our final purpose is to create a device that can allow the IoT and the Big Internet to communicate with each other seamlessly. An open standard that perfectly suits the project needs is 6LoWPAN.

5.1 6LoWPAN

6LoWPAN[20], acronym for IPv6 over Low power Wireless Personal Area Networks, is a technology that uses IPv6 to allow small devices to communicate with each other.

6LoWPAN implements the IEEE 802.15.4 standard, providing, on top of it, IPv6 support. The decision of creating a specification to use IP on these networks was driven by the following:

- IPv6 is auto-configurable
- IPv6 is stateless
- As LoWPANs are normally deployed with a large number of devices, a large address space (like the one provided in IPv6) is needed
- IPv6 makes easier to connect LoWPANs to other IP networks, like the Big Internet.

IEEE 802.15.4 allows the use of either 64-bit extended addresses or 16-bit addresses (unique within the PAN). 6LoWPAN supports both types of addresses, and enforce further constaints on 16-bit addresses, reserving some patterns. To allow IPv6 packets to be carried from over IEEE 802.15.4 based networks, 6LoWPAN defines a header compression mechanism. Furthermore, the MTU size for IPv6 packets over IEEE 802.15.4 is 1280 octets but, as shown in section , the maximum frame size at the media access control layer is 102 octets. As a solution, 6LoWPAN provides a fragmention and reassembly adaptation layer below IPv6.

5.2 6LoWPAN frame format

In 6LoWPAN, every PDU is composed of an incapsulation headers followed by the payload. These headers are staked and contains data about, in order:

1. Mesh addressing
2. Hop-by-hop options
3. Fragmentation
4. Payload

Each header in the header stack contains a header type followed by zero or more header fields.

These headers can be of four different types:

- NALP, identified by
- Dispatch, identified by a 0 in the first bit and a 1 in the second bit
- Mesh Addressing
- Fragmentation

NALP (Not a LoWPAN frame) specifies that the following bits are not a part of the LoWPAN encapsulation, and, if encountered by any LoWPAN node, the packet containing such header should be discarded.

The Dispatch header type contains, after the first two bits, a 6 bits selector. The selector further indicates the type of dispatch header. The actual header follows the selector immediately. The dispatch header is itself classified under different types:

- IPv6: Indicates an uncomperssed IPv6 header
- LOWPAN_HC1: Indicates a LOWPAN_HC1 compressed IPv6 header

- LOWPAN_BC0: Indicates a LOWPAN_BC0 header for mesh broadcast/multicast support
- ESC: Indicates that the header a single 8-bit field for the Dispatch value.

The Mesh Addressing header are composed of the following fields:

- V: a 1-bit field that indicates that the Originator Address is an IEEE extended 64-bit address if set to 0, a short 16-bit addresses if set to 1
- F: a 1-bit field that works as the V field except it specifies the type of address of the final destination of the packet.
- Hops Left: a This 4-bit field representing the number of hopes left for the packed. It is decreased by 1 at every hop.
- Originator Address: a field representing the link-layer address of the Originator
- Final Destination Address: link-layer address of the Final Destination of the packet.

The Fragmentation header contains the information to reassemble a fragmented datagram that doesn't fit within a single frame. Fragmentation header is composed of the following fields:

- datagram_size: a 11-bit field that encodes the size of the entire packet (before fragmentation)
- datagram_tag: uniquely identifies the datagram
- datagram_offset: present only in the second and subsequent link fragments, specifies the offset of the fragment, in bytes, from the beginning of the datagram.

5.3 REST protocols

Devices in the IoT, using 6LoWPAN standard, are able to exchange IPv6 packets. To allow a simple way to develop IoT services, we have to use a good application layer. Recent work in IoT standards, has been oriented by the example of the Internet. The majority of the Internet works with IP, and is mostly used to carry HTTP datagrams. HTTP 1.1 (HyperText Transfer

Protocol) is based on the definition of ”REST” architecture, using, as the basic data being shared, hypertext documents. REST (REpresentational State Transfer) architecture, introduced in 2000 by Roy Fielding[15] for use on distributed systems such as the World Wide Web, defines a set of principles and constraints to make client-server stateless communications.

The central principle of REST is the existence of resources referenced with a global identifier. Each objects in the network can communicate with another by knowing only the identifier of the desired resource and action. There can be any number of other network components between the two objects, which are completely unaware of the path taken by the message being transferred. Each resource can be presented, in response to requests, in different representations.

Network systems, in order to be called REST, must comply with the following constraints:

1. Client–server: clients and server are separated by a uniform interface
2. Stateless: no client context can be stored on servers. Servers can have states, that must reachable and identified like any other resource
3. Cacheable: REST protocols must define a way to control caching of resources to reduce the number of interactions between components. Network objects must also be allowed to indicate the caching properties of a certain resource to prevent clients from using exipired, and therefore inappropriate, data
4. Layered system: clients are unaware of the path being taken by their requests
5. Uniform interface: the uniform interface between clients and server, must be created following these principles:
    - Identification of resources: resources are individually identified and can be accessed though their representations
    - Manipulation of a resource using its representation: clients, if allowed, are able to modify or delete the resource on the server
    - Self-descriptive messages: messages carry within themselves all the necessary data to be processed
6. Code on demand: optionally, in a REST architecture, servers should be able to extend clients functionalities transferring executable code.

To bring REST architectures to the IoT, a protocol specifically thought for LoWPANs have been recently defined: CoAP.

Chapter 6

Constrained Application Protocol: CoAP

CoAP is a web protocol designed thinking at the special requirements of those constrained environments typical of the IoT networks. The protocol realizes a subset of REST, common with HTTP, optimized for "machine to machine" applications, plus some extra features like:

- Built-in discovery
- Multicast support
- Asynchronous message exchanges.

The CoAP communication model is similar to the client/server paradigm in HTTP, with end points requesting actions on resources. However, CoAP messages are exchanged asynchronously over a datagram-oriented transport (UDP).

CoAP defines 4 kind of messages:

- Confirmable (CON)
- Non-Confirmable (NON)
- Acknowledgement (ACK)
- Reset (RST)

Confirmable messages and acknowledgements are used to provide reliability to a message exchange. When a CoAP end point sends a confirmable message to another device, it waits for a default amount of time to receive a corresponding acknowledgement message. If the time expires, the message is sent again. If the recipient is not able to process the confirmable message, it sends a reset message instead of an ACK.

6.1 Requests and responses

Using CoAP messages, devices can execute requests and receive responses. Requests can be confirmable and non-confirmable. If a request is confirmable, and the receiver can respond immediately, the response content is carried directly in the acknowledgement message. This message is called a "piggy-backed" response. If the receiver cannot respond immediately, it sends an ACK to the sender, so that it can stop retransmitting the request. When the content is ready, the receiver sends it to the client in a confirmable message, expecting an acknowledgement in return. As congestion control mechanism, whenever a timeout is triggered, the end point waiting for the acknowledgement will set the timeout time on the request to the double of the current value.

6.2 CoAP message format

CoAP messages are encoded in binary form. Every message consists of three parts:

- The header
- The options section
- The payload

The header section consists of the following:

- Version: a 2-bit unsigned integer that indicates the CoAP version number.
- Type: a 2-bit unsigned integer that indicates the message type. Valid values are:
  - 0 (CON)
  - 1 (NON)
  - 2 (ACK)
  - 3 (RST)

- Option Count: a 4-bit unsigned integer that indicates the number of options after the header. If set to zero, indicates that there are no options and that the payload follows the header immediately.

- Code: an 8-bit unsigned integer that, if the message is a request, indicates the request method and, if the message is a response,represents the response code.

- Message ID: a 16-bit unsigned integer that identifies a message, to avoid duplication and to match CON messages and ACK/RST messages.

The options in a CoAP message appear ordered by their option number. The option number for each option is calculated with a delta encoding: the delta field of the current option is summed to the option number of the previous option, if any, zero otherwise.

The fields in an option are defined as follows:

- Option Delta: 4-bit unsigned integer that indicates the difference between the option number of this option and the option number of the previous option (or zero, if it's the first option). The Option Numbers 14, 28, 42 are reserved for empty options, which are ignored and are used in case a delta larger than 15 is needed

- Length: Indicates, in bytes, the length of the Option Value. By default, Length is a 4-bit unsigned integer, but if the Option Value is larger, Length can be extended with an extra 8-bit integer. If the extra integer is needed, Length value is set to 15 and the value of the extra integer is added to this number, allowing a maximum length of 270 bytes for Option Value.

6.3 CoAP Requests

As explained in section 6.2, in CoAP a message of type Request has its method indicated in the Code section of its header. CoAP Requests, as for the draft draft-ietf-core-coap-08[23], can be only one of the following methods:

- GET

- POST

- PUT

- DELETE.

The method GET is “safe”, meaning that a GET request can only retrieve a resource. Furthermore, GET, PUT and DELETE are “idempotent”, meaning that they have the same effect even if they are invoked several times.

Safe and idempotent derive directly from HTTP 1.1[14].

6.4 CoAP Responses

Responses in CoAP are identified by the Code field (that indicates the response code) in the header. Response codes are used to indicate to the client the result of the attempt to satisfy the request. As explained in section “CoAP message format” (6.2), the Code field is 8-bit long. The upper 3 bits indicates the class of the response code. CoAP response classes are:

- 2 - Success, indicates that the request was successfully handled
- 4 - Client Error, tells the client that the request contains bad syntax or cannot be fulfilled in the form it was sent
- 5 - Server Error, used when the server fails to fulfill an (apparently) valid request.

The lower 5 bytes provide additional detail about the response.

6.5 CoAP Options

CoAP options fall into one of two classes: “critical” or “elective”. Critical options have an odd Option Number, whereas elective options have an even Option Number. Elective and critical options are handled differently:

- An unrecognized option of class “elective” must be silently ignored
- An unrecognized option of class “critical” that occur in a CON request,must cause the return of a 4.02 (Bad Option) response
- If an unrecognized critical option occur in a confirmable response, the response should be rejected with a reset message
- If an unrecognized critical option occur in a non-confirmable, response or request, the message is silently ignored.

CoAP options set is shared between requests and responses. The options defined by default in the protocol are the following:

- Content-Type (critical): Indicates the format of the message payload
- ETag (elective): Represents the entity-tag of a resource. In a response, identifies the value of the resource representation. In a request, is used to interrogate the server to establish whether the resource already possessed by the client is still valid or needs to be refreshed.
- Location-Path and Location-Query (elective): Indicate the new location of a resource created with a POST request
- Max-Age (elective): Indicates the maximum time the response can still be considered "fresh" and be served by a cache
- Proxy-Uri (critical): Specifies the URI of a remote resource. Indicates that the receiving end point should act as proxy and forward the request, decoding the Proxy-Uri option value and splitting it in Uri-Host, Uri-Port, Uri-Path and Uri-Query. Can occur more than once and, in that case, the final URI of the proxy is retrieved concatenating all Proxy-Uri options values
- Token (critical): Is used to match a response with a request. Every request has a client generated token which the server echoes in the response. Messages should be identified by the pair Token-client, for servers, and Token-server, for clients.
- Uri-Host (critical): Specifies the Internet host of the resource being requested
- Uri-Port (critical): Specifies the port number of the resource
- Uri-Path (critical): Specifies one segment of the resource's path and may occur more than once
- Uri-Query (critical): Specifies one argument parameterizing the resource.
- Accept (elective): Is used by the client to indicate the media types that are acceptable in the response
- If-Match (critical): Is used to perform a "conditional" request for a resource. If-Match option value can be either an ETag, indicating a representation of the requested resource, or an empty string. An

empty string places the precondition of the existence of any representation for the resource

- If-None-Match (critical): As opposed to If-Match, this option make a request conditional on the non-existence of the target resource. If-None-Match values are ETags.

6.6 CoAP Blockwise Transfer

As 802.15.4 based networks standard packet packet size is 127 octets, and that the available space is further reduced by the overhead introduced by 6LoWPAN and UDP headers, CoAP nodes may need to use packet fragmentation to exchange large amounts of data, for example for firmware updates or to advertise their resources using CoAP link format.

To allow application layer fragmentation in CoAP, two options are used: Block1, that is used to control a blockwise request, useful for methods such as PUT and POST, and Block2, which instead pertains to the response payload.

- NUM: a field of variable size (4, 12, or 20 bit) that contains an unsigned integer that indicates the block number. Block number 0 indicates the first block.

- M: a (1 bit) flag that indicates whether there are more additional blocks available or not

- SZX: a 3 bit field that indicates the size of the block to the power of two.

Block1 and Block2 can be used in two different ways: to describe a blockwise transfer and to control a blockwise retrieval. The role of "controller" options is taken the two options are present on packets going in the opposite direction (i.e. Block1 in a response and Block2 in a request).

Chapter 7

System requirements, constraints and architecture

7.1 Project features

The purpose of the project is to create a proxy that can translate CoAP into HTTP and vice versa, in order to create a connection between the Internet of things and the big Internet. A a starting base, the proxy will have to provide the following features:

- Handle requests and responses in bot CoAP and HTTP protocols
- Map HTTP URIs into CoAP ones and vice versa
- Map Content-Types from one protocol to another, whenever possible, and report unsatisfiable requests adequately
- Store responses with a caching system and use them to serve data faster and at a lower cost.

Besides providing these feature, the proxy must be design to meet some typical software requirements:

- Modularity: although the proxy is meant to handle only HTTP-CoAP requests and responses mapping, it must be designed to allow easy and seamless integration of additional protocol translation features
- Robustness: the proxy must be able to sustain faults. Furthermore, whenever a crash occurs while performing some operation over a single protocol, the proxy should be able to continue handling requests and responses over the remaining ones. At the same time, the system should try to restore its functions over the crashed protocol.

7.2 Physical constraints

The most restrictive constraints that must be taken into account are enforced by the devices the proxy will run on. As a matter of fact, the proxy will probably run on devices with limited memory size and computational power. Furthermore, on these machines, the storage devices could be unable to sustain big amounts of writes. To be usable on these kinds of devices, the proxy should be designed with the following features:

- Low power usage: the program will have to save as many CPU cycles as possible
- Low memory usage
- Low storage usage (as few writes as possible).

The physical constraints enforced will have to be taken into account especially while choosing the development instruments.

7.3 High level design

7.4 What we want to accomplish

Although the described standards are designed to allow straightforward mapping of IoT networks communication onto the Big Internet, integrating the two systems is all but a trivial tasks. The mapping process must apply to all the level of the stack, and the communication must be allowed in both ways.

Aside from the problematics deriving from the mapping itself, we must take into account that the two technologies, the IoT and Internet, are designed to run on network devices that have a huge gap in terms of computational and communication capabilities.

In this chapter, we will discuss the design of a proxy that can seamlessly map a IoT request to be sent through the Internet and vice-versa. We will first subdivide the tasks of the proxy into functional subparts and analyze the high level design of the software component. We will then discuss how to handle the communication between the proxy subcomponents. Finally, we will choose the tools to use for the proxy implementation, choosing them basing on how they adapt to our design choices.

The project takes the name of Kink.

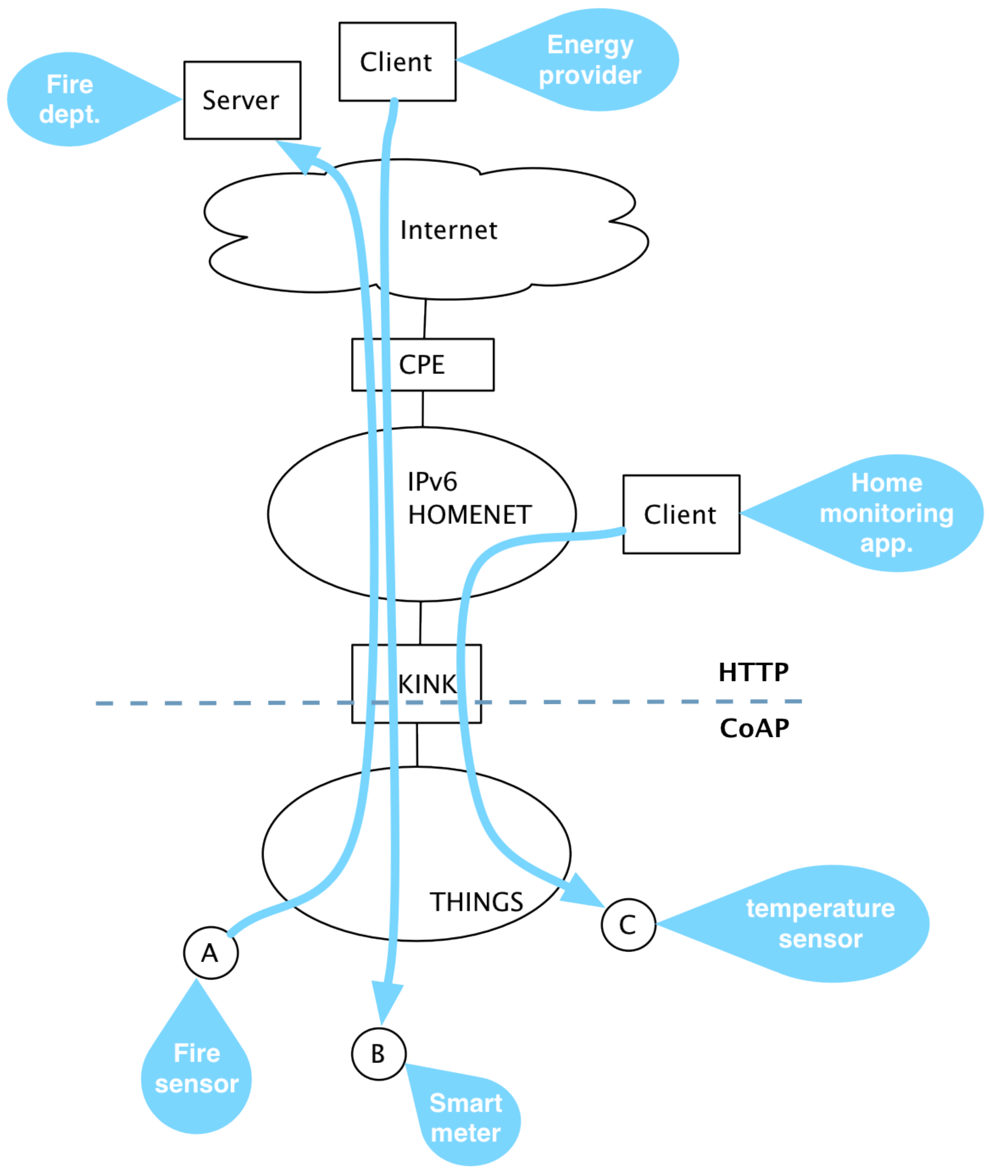


Figure 7.1: Kink serves as a proxy between the IoT and the Internet

7.5 The design process

The first step of the design process is to subdivide the proxy into some subcomponents in order to guarantee the modularity of the system and to make the designing process easier. In order to do so, the requirements must be further analyzed to find those features that can be grouped and made into a module. Once the main parts of the proxy are defined, the design process should proceed by analyzing how the components should communicate among each other. The most evident subgroups of features are those that are related to a single protocol. The proxy will need two distinct components that:

- Handle requests and responses in HTTP
- Handle requests and responses in CoAP.

The components above are needed for the first step of the presented use case scenario: an HTTP request is received by the proxy, more specifically, by its component that can handle the HTTP protocol. In the second step, the HTTP handler must forward the request to another component that has access to the queried resource. In order to do so, a protocol handler module must be aware of the presence of all the other protocol components. Furthermore, the handler must be able to find the component that has access to the resource indicated in the request it received. As this last feature is common to every protocol component, it can be provided by a separate module. This new component will take care of the routing and URI mapping procedures. The HTTP component can now interrogate the URI mapper/router in order to find the protocol handler to which it must forward the request. This operation must be performed taking into account that:

- The forwarded request must be in a format that the receiving protocol handler can understand. This can be achieved by designing a common format for requests and responses or, in this case, by providing the CoAP handler with HTTP to CoAP request translation feature
- To avoid unnecessary message exchanges, the HTTP component should include in the request the mapped URI of the CoAP resource needed. This way, the CoAP handler can handle the request without querying the URI mapper.

Now, the CoAP component has a request (translated) to handle, and it must treat it as a normal CoAP request. Acting as a client, the handler sends the request to the appropriate end point. When the CoAP component receives the response, it must send it back to the HTTP handler. In order to do so, it must know from which protocol handler the request came from. This can be simply achieved by adding the back route to the request sent to the CoAP handler. The HTTP handler must now translate the response in HTTP and send it back to the corresponding client.

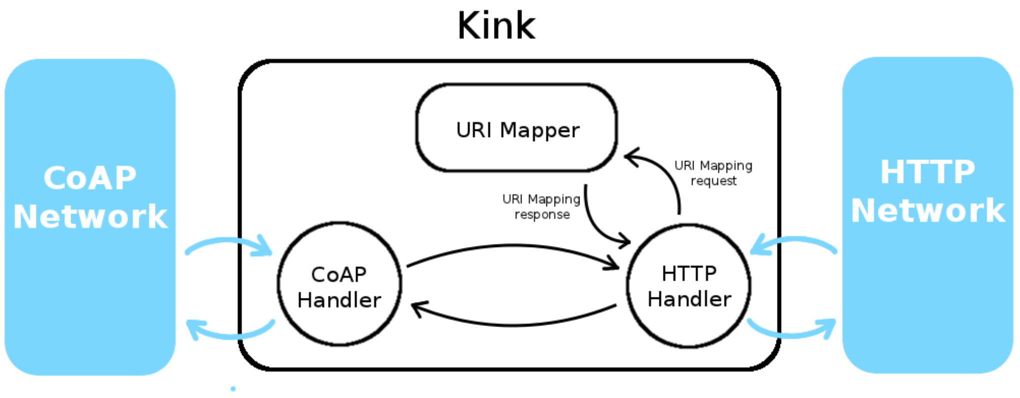


Figure 7.2: Kink high level design: a request coming from an HTTP network is received by the corresponding protocol handler. The HTTP handler interrogates the URI mapper component, and forwards the message to the CoAP handler. The request is than mapped to CoAP and sent to the CoAP network.

The flow described above can now be summarized with:

- The HTTP handler receives the request
- The HTTP handler queries the URI mapper/router to retrieve:
  - The mapped URI
  - The route to the handler that has access to the resource
- The HTTP handler saves the request and forwards it to the CoAP handler, adding the required information about the back route
- The CoAP handler translates the request and, using the mapped URI, sends it to the CoAP end point with the requested resource
- The CoAP handler receives the response from the end point and sends it back to the HTTP handler through the indicated back route

- The HTTP handler can now translate the response and send it back to the original client (which it can find using the stored request informations)

The basic components and their behavior are now defined. Before proceeding further, the communication system among the described modules must be better defined.

7.6 Low level design: communication among components

All the proxy components must be provided with channels to communicate among each other. These channels must be shared among components in an easy. Furthermore, exchanged messages, until they are consumed by a handler, must persist on the system, ideally, even in case of a proxy crash or a system reboot.

As we are working on a constrained environment, we don't want the proxy to be heavily multithreaded (e.g. using a thread-per-request policy). Request should be handled by a single process one after another, and the other incoming messages must be stored in a suitable data structure: a (possibly bounded) queue.

As emerges by the considerations made above, a Kink protocol handler will have the following communication channels:

- Listening sockets, to handle incoming requests
- Connected sockets, to connect to protocol specific servers on behalf of other Kink protocol handlers
- A channel that allows RPC communication with the Kink process that handles the core features, like URI mapping
- An internal queue to store incoming messages.

7.7 Mapping CoAP features

Before choosing the tools that will be used to implement the proxy, a closer look to the protocols is needed. Some CoAP features can be difficult to translate in HTTP, and need to be deeper analyzed before taking more design choices.

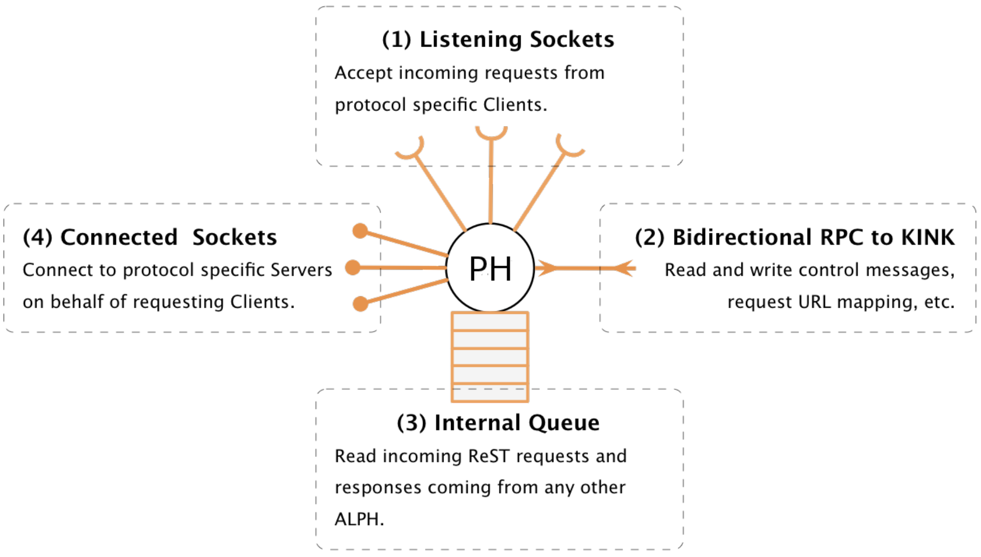

(1) Listening Sockets
Accept incoming requests from protocol specific Clients.
(4) Connected Sockets
Connect to protocol specific Servers on behalf of requesting Clients.
PH
(2) Bidirectional RPC to KINK
Read and write control messages, request URL mapping, etc.
(3) Internal Queue
Read incoming ReST requests and responses coming from any other ALPH.

Figure 7.3: A Kink protocol handler, with its in/out channels.

7.8 Content type mapping

Allowed content-types in CoAP are:

- text/plain
- application/link-format
- application/xml
- application/octet-stream
- application/exi
- application/json

To any other content-type request coming from an HTTP end point and directed to a CoAP resource, the Proxy must respond with a response with code HTTP 415 "Unsupported Media Type" When a CoAP client performs a requests for content-type application/link-format to an HTTP servers, it must reveive a response with code CoAP 4.15 "Unsupported Media Type".

7.9 URI mapping

The mapping of a CoAP URI to an HTTP URI and vice versa, can be done statically or dinamically. Static URI mapping can be performed in two ways:

- Homogeneous mapping
- Embedded mapping

These methods are discussed in the following subsections.

Homogeneous mapping

The mapping between CoAP and HTTP URIs is homogeneous when the same resource is identified by URIs that differ only by their schemas.

For example, a URI like http://www.example.com/coapresource is mapped with coap://www.example.com/coapresource .

Embedded mapping

The mapping is said to be embedded when the authority and path of the native URI being mapped are included in the resulting URI.

For example, the URI coap://coap.sensornet.org/coapresource is mapped with http://proxy.httpcoap.net/coap/coap.sensornet.org/coapresource.

Multicast

On CoAP, multicast requests can be performed[23]. These requests need special treatment in order to be translated from and to HTTP. In order to successfully handle a multicast request, the proxy must:

- Establish whether the requested URI identifies a group of nodes.
- Map the request, distributing it to the involved end points.

The request distribution operation consists of the following steps:

- The CoAP component sends out the request to the group of servers
- The handler collects the responses (until a timeout occurs)
- The proxy translates the responses in HTTP and sends it back to the client.

Multipart media type can be used to translate the set of received responses to HTTP[14]. A simple approach is to map each response from CoAP to HTTP. Then all the responses, represented with "message/http" media type, are delivered in a single HTTP response with "multipart/mixed" media type.

Observe

CoAP offers a subscription feature called "CoAP observe"[16]. A client interested in a resource can send an observe request to an end point. This end point will then, while the subscription is still active, send a message to the client containing the resource anytime a certain event occurs (e.g. the resource changes). A subscription ends whenever:

- The client stops sending acknowledgement messages to the server, or
- The server stops sending messages to the client.

There are two different HTTP features[18] that can be used to map a CoAP subscription:

- Streaming: works in a way that's more similar to CoAP observe: the connection is kept and a message is sent to the client every time there's a data update
- Long-polling: when it receives a long-poll request, the server keeps it until the requested resource is ready. The client waits until either timeout occurs or it receives the response from the server, then it immediately sends another request.

Although streaming may look like a better option, long-polling is probably the best choice for CoAP subscription mapping in HTTP:

- With long-polling, after every event, the client has to send a new request in order to retrieve more data. However, this shouldn't cause any latency, as it's unlikely that the server will have new data to send to the client before it receives the new request.
- With streaming, the server must keep all the connections with the subscribed clients indefinitely, even if they are not interested in the resource anymore.

A considerable issue of long-polling is that whenever a resource is ready, it is sent to all the interested clients at the same time. The clients that are still interested will send a new request to the server, resulting in many requests sent simultaneously, with a big impact on the proxy performances. Also, for every new long-polling request, the proxy will have to perform a TCP connection setup/tear-down. However, the cost of the connection can be minimized using HTTP keep-alive.

Link format and resource discovery

CoAP defines a link format[22] to be used by server devices to describe their available resources. The CoAP Link Format, like for HTTP Web Linking, is carried as a payload. The link format is reachable at a well-known URI (by default, /.well-known/core) and is assigned a media type. CoAP link format can be used for resource discovery using a multicast GET request on the well-known URI for all the connected nodes.

Each link in CoAP link format presents an URI which represents a resource. Additional information can be provided using the following attributes:

- Resource Type (rt) attribute is used to describe the resource
- Interface Description (if) attribute indicates how the resource can be used
- Maximum size estimate (sz) gives an estimation of the size of the data being stored at the given location.

These attributes can also be used in the request query to filter on the resources being displayed in the link format description.

In CoAP link format additional informations about relationships between resources can be represented using the anchor attribute.

We will now present some examples of link format requests and responses. The first example is a GET request to /.well-known/core, to which corresponds a response with two resources represented: a temperature sensor and a light sensor. Additional information about the sensors, particularly their location, is provided by the rt attribute.

```
REQ: GET /.well-known/core

RES: 2.05 "Content"
   </sensors/temp>;if="sensor";rt="OutTemp",
   </sensors/light>;if="sensor";rt="OutLight"
```

In the next example, using the anchor attribute, the server indicates an alternative (shorter) URI for the light sensor.

```
REQ: GET /.well-known/core

RES: 2.05 "Content"
   </sensors/temp>;rt="OutTemp";if="sensor",
   </sensors/light>;rt="OutLight";if="sensor",
   </l>;anchor="/sensors/light";rel="alternate"
```

In the last example a more complex query filters on the resource type, requesting only the resource representing an external temperature sensor.

```
REQ: GET /.well-known/core?rt=OutTemp

RES: 2.05 "Content"
   </sensors/temp>;rt="OutTemp";if="sensor",
```

CoAP link format will be used in the final implementation of the proxy as the simplest (and default) way to perform resource discovery over devices connected in the CoAP network.

7.10 Development tools

Before implementing the proxy, we must choose the right development instruments. These instruments must fit the project needs, respect the constraints and perform as well as possible.

Tools features

The number of the lines of code needed for the project can be reduced by using some tools that provide some usefull features. In particular, the most important features are:

- A CoAP protocol implementation
- A HTTP protocol implementation
- DNS support
- Long polling support
- An extensible cache component, or the libraries to build one

Constraints

All the hardware and software constraints are to be taken into account while choosing the tools. On top of them, one more constraint is enforced: the tools must be registered under a suitable license, as the final product will have to be completely free and open source.

Evaluation parameters

The tools that pass all the constraints checks provide, must then to be evaluated in order to find the best choice. The tools evaluation is performed using, as a starting base, the following parameters:

- Estimated lines of code
- Resources consumption: the final product must be as optimized as possible
- Estimated performance
- Robustness, maturity and support
- Efforts needed to adapt the tools to the design.

The first choice to be made is between two different approaches: extend an existing product or assemble a new one.

Extension: Apache

Apache is a web server initially developed in 1995[1]. Apache is a project that is part of the Apache Software Foundation, and is developed and maintained by an open community of volunteer developers located around the world.

The Apache project is mature and robust and is well maintained and supported. Apache 2.0 provides a valid and tested HTTP implementation, fitting one of the project needs. Being a web server, Apache is already optimized to handle a huge number of HTTP requests. Furthermore, Apache is easily extensible through a simple module system. However, this web server presents some issues:

- Being a mature project, thought for the web, Apache has an enormous list of features that are not needed for the proxy project and that may in fact cause:

  - a loss in performance in the final product
  - confusion in the project documentation
  - unexpected issues while interacting with other extensions

- Apache uses a "one thread per request" policy, which may lead to devastating performance drops when used to serve long poll requests

- Module extension documentation is out of date[17] and the users are warned about it being unreliable[9].

Extension: Varnish

Varnish is an HTTP accelerator with 5+ years on the trenches of web sites such as Facebook and Globo. Varnish can be installed in front of any kind of web application. speeding them up by caching the incoming requests. The first version of the project was released in 2006, and has grown to become a mature product in the next years. Varnish is developed and maintained by a company named "Varnish Software" and is released under the BSD license[4]. Even though Varnish is a free and open source software, users can purchase enterprise subscriptions, to get professional assistance by the project developers.

Varnish can be extended with new modules. Although this feature lacks extensive documentation, example modules can be found on the project website[5], and can be used as a starting base for modules developers.

Varnish could be a good starting point for the HTTP/CoAP proxy, as it provides a tested HTTP protocol implementation and an optimized caching system, however, like Apache, it presents some issues:

- As Varnish was designed to work on performant hardware, it may not be suitable for our needs

- Varnish is heavily threaded and, like on Apache, a separate worker thread handles each incoming client connection. The number of worker threads is limited, and when the limit is reached incoming requests are added to a queue. This queue is also limited, and exceeding connections are rejected. As with Apache, this may cause performance drops when serving long poll requests

Extension: Squid 3

Squid 3 is a proxy server and web cache daemon. It is rich of features and well tested under incredibly heavy loads (e.g. it provides the frontend for Wikipedia's entire public infrastructure). Unlike Varnish, Squid has no plugin interface, and needs some quite intrusive tweaks to handle the CoAP bits. Namely, one has to heavily patch the FwdState and HttpStateData classes for handling the protocol flow and needed translations; furthermore, the cache system has to be extended to implement CoAP's caching and freshness maintenance policies. Additionally, the same overall "defect" of Apache and Varnish was found also in Squid: the general purpose nature of the software architecture leads to inflation in the memory footprint, making it unfit for the embedded nature of the KINK target.

Assemble: libevent

Libevent is a library created to simplify the development of event driven applications. Libevent is distributed under an open source license, and includes some useful features, like an HTTP implementation and DNS support. The libevent library is mature and well supported and documented. Furthermore, "event driven architecture" fits our needs, as it reduces CPU usage by relieving the proxy handlers from polling over connections.

Assemble: libev + libebb

Libev, in some way, is an evolution of libevent. It provides the same features of libevent's core, but performs slightly better[3]. Unlike libevent, libev lacks HTTP and DNS support. A project called libebb adds HTTP support to libev, but it's not mature enough to guarantee the robustness needed for the HTTP/CoAP project [8]. As an alternative solution, libev offers an emulation layer that provides interfaces on the DNS and HTTP components of libevent.

CoAP implementations

All the tools mentioned above lack CoAP support. To simplify the proxy implementation process, we need to find a suitable library that can easily be used to extend the tools of choice with all the CoAP protocol features.

libcoap
Libcoap is a CoAP protocol implementation, written in c, developed by Olaf Bergmann[12]. Libcoap is is free and is licensed under the GPL 2.0 license[6].

evcoap

Evcoap is a CoAP implementation that is designed to be the CoAP equivalent of the HTTP implementation of libevent. Evcoap is an open-source product written by KoanLogic, and is currently in development. Evcoap supports blockwise transfer and observe, which are two very important feature for any CoAP based application. The evcoap module fully implements the CoAP protocol as per draft-ietf-core-coap-08 with server, client and proxy roles.

Other tools

POSIX message queues

POSIX message queues allow unix processes to exchange messages. POSIX mqueues are a perfect tool to create the message channel among the proxy protocol handlers, as they are persistent (messages are kept even in case of system reboot) and portable on any POSIX system.

LibU

LibU is a library that includes modules for several tasks like:

- Store data in an hashmap (useful for the cache implementation)
- Memory allocation
- Testing
- Debugging and logging.

LibU comes under a BSD-style license, and can be used and modified freely.

Conclusions

As emerges from the above analysis, the following tools combined are the most suitable to the project needs: libevent paired with evcoap. These tools provide most of the necessary features and are sufficiently robust and performant. Along these instruments, we will use the POSIX message queues as a communication channel among the proxy components and LibU as a general purpose library.

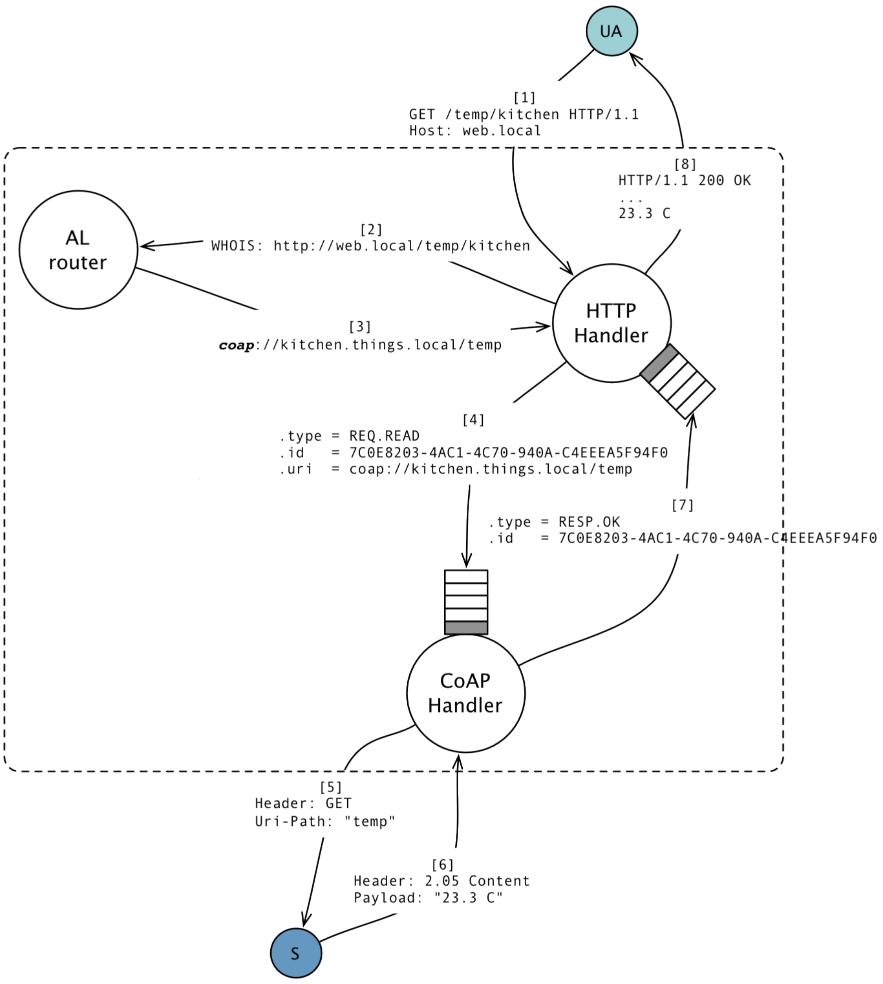

UA
[1]
GET /temp/kitchen HTTP/1.1
Host: web.local
[8]
HTTP/1.1 200 OK
...
23.3 C
AL router
[2]
WHOIS: http://web.local/temp/kitchen
HTTP Handler
[3]
coap://kitchen.things.local/temp
[4]
.type = REQ.READ
.id = 7C0E8203-4AC1-4C70-940A-C4EEEA5F94F0
.uri = coap://kitchen.things.local/temp
[7]
.type = RESP.OK
.id = 7C0E8203-4AC1-4C70-940A-C4EEEA5F94F0
CoAP Handler
[5]
Header: GET
Uri-Path: "temp"
[6]
Header: 2.05 Content
Payload: "23.3 C"
S

Figure 7.4: The final Kink design, with an example of the flow of a request from HTTP to CoAP.

Chapter 8

Designing a cache for the proxy

8.1 Introduction

Devices that uses CoAP as their communication protocol usually have low computational power. Also, CoAP devices work in constrained environments where power consumption is a vital factor. Some devices might even be asleep the majority of the time in order to save as much power as possible.

Under these conditions, a caching system is a must in a CoAP/HTTP mapping proxy.

8.2 Requirements

A CoAP/HTTP proxy is meant to perform translation of messages between REST protocols. As caching principles are similar among REST protocols, we will use their typical strategies as a starting base for our system. The rules that will be taken as a starting point are:

- Whenever a request is received, if it's cacheable, the proxy cache performs a lookup in order to find the corresponding response

- A cache entry must validate in order to be sent back to the client. Validation policies may vary depending on the protocol that serves the requested resource

- A protocol may offer a revalidation system (e.g. based on ETags)

- The cache must store statistical data about a the popularity of any cached resource (e. g. how many times in a certain period of time the resource has been requested).

Since these simple principles can be applied to both CoAP and HTTP, but we want our system to be as flexible as possible, we must design it to grant room for protocol specific policies. In the next section8.3, CoAP's caching policies will be presented.

8.3 Caching in CoAP

We will now review the core concepts concepts given in section8.2, describing how they apply to CoAP.

Matching a request in cache

In CoAP, a request matches another if and only if they have the same method and options.

Cacheability of CoAP responses

In CoAP, the cacheability of responses does not depend on the request method. The caching system must look for the response code in order to establish whether they are cacheable or not. Cacheability based on response codes can be summarized with these rules:

- none of the response with 2.xx (Success) code are cacheable except:
  - 2.03 (Valid, used for Etag revalidation of responses)
  - 2.05 (Content)
- all the responses with 4.xx (Client Error) code are cacheable, but cannot be validated
- all the responses with 5.xx (Server Error) code are cacheable, but cannot be validated

Freshness

CoAP uses a a method, borrowed by HTTP/1.1, called "freshness model" to define its caching policies. A matched cache entry can be served as a response if it's fresh. The cache uses two different parameters to establish whether an entry is stale or fresh:

- The response "max-age" option, stored on a previous request;
- A timestamp of the last time the response has been served.

"Max-age" option defaults to 60 seconds. A cached response is considered fresh if $RT < CRT + CRMA$, where RT is the time the new request was sent to the server, CRT is the time the cached response was last served and CRMA is its max-age.

Validation

CoAP offers a revalidation system based on ETags[14]. When a cache entry is stale, the proxy can include the Etag of the stored response in the new request. The endpoint can tell, in its response, if the stored value is still valid, or send a new value to replace the old one.

Optimizations

Sometimes, we can save some messages by subscribing[16] to a resource that is too often requested and found stale. This kind of optimizations can be done by a separate process that acts as "supervisor" of the cache. This process takes care of collecting data about the resources being requested and tries to reduce the number of messages between the proxy and the CoAP net.

8.4 Integration of the cache system in the overall design

The design of the proxy cache must take into account the whole structure of the system. As seen in the previous chapters, the proxy system, to guarantee robustness and flexibility, contains as many modules as the protocols it handles. All those modules are pieces of code that run in separate processes.

Protocols also have different policies regarding their cache systems, although it's possible to identify some common features. Two different approaches can be taken in designing the cache system: adding caching features to protocol handlers, or create a centralized cache handler, running on a separate process, that stores requests for all protocols. Both approaches have upsides and downsides.

A cache handler designed as a separate entity would have the following features:

- A communication protocol between protocol handler processes and the cache system
- A supervisor process, extended with per-protocol features, that takes care of optimization
- A system to store different kind of requests.

On the other hand, cache functions added to protocol handler would be designed to include the following features:

- A module implementing common cache behavior, shared by all protocols;
- Per protocol extensions that take specific care of protocol properties and optimizations
- An optional communication system between cache handlers, again for optimization purpose.

A centralized cache would use less memory. Also, as all cache entries are stored and handled by the same process, per protocol optimizations are easier to implement: no IPC between protocols is needed. On the other hand, protocol handlers with cache features would perform better, as memory access is much faster than IPC. Thorough analysis of all these facets will be given in the next section8.5.

8.5 Use case scenarios

Before proceeding further, we need to take into account some more specific use case scenarios. In the first use case scenario, two protocol handlers act as server instances, and they both query a third handler for the same resource. We want the queried handler to respond with a cached response at least to the second request. This is no trivial task, as the following issues are to be considered:

- In a decentralized cache, the issues shown by this scenario can be solved by saving the responses in those protocol handlers that act as clients. This design may reduce performances, as a request must be translated and passed between protocol handlers even if the response

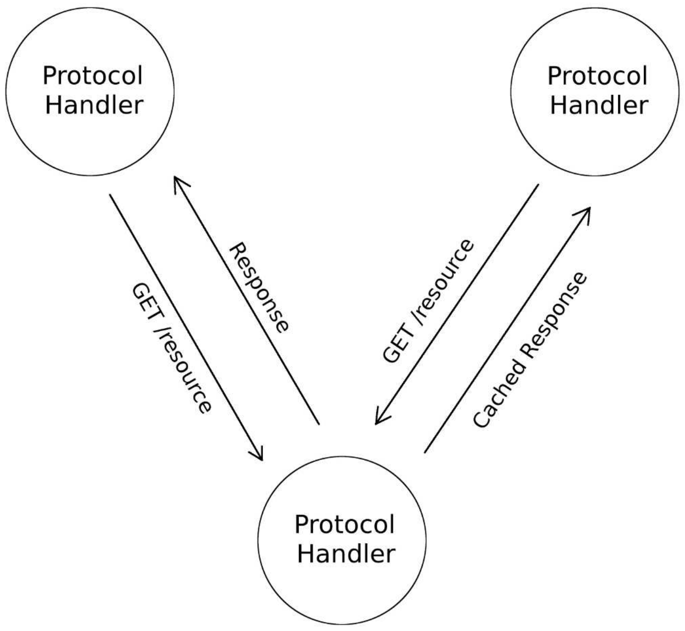


Figure 8.1: Cache use case scenario 1: two protocol handler attempting to access the same resource on a third

is already cached. To improve performance, and fully take advantage of the cache system, already translated responses may be stored also on server-side protocol handlers. This approach would however cause more memory usage and data redundancy

- In a centralized cache the same issues can be solved by storing all responses in a common format, leaving to the protocol handlers the task to retranslate them in their native format. The performance loss due to the translation procedure can be avoided by saving each response already translated in the protocol used by each protocol handler, causing the same issues presented in the previous point.

In the next use case scenario an example of per protocol optimization is presented. As previously shown, the CoAP observe feature can be used by the proxy to establish a subscription to a frequently requested resource in order to reduce the number of messages exchanged. Designing this feature can be more or less troublesome, depending of the cache design we choose.

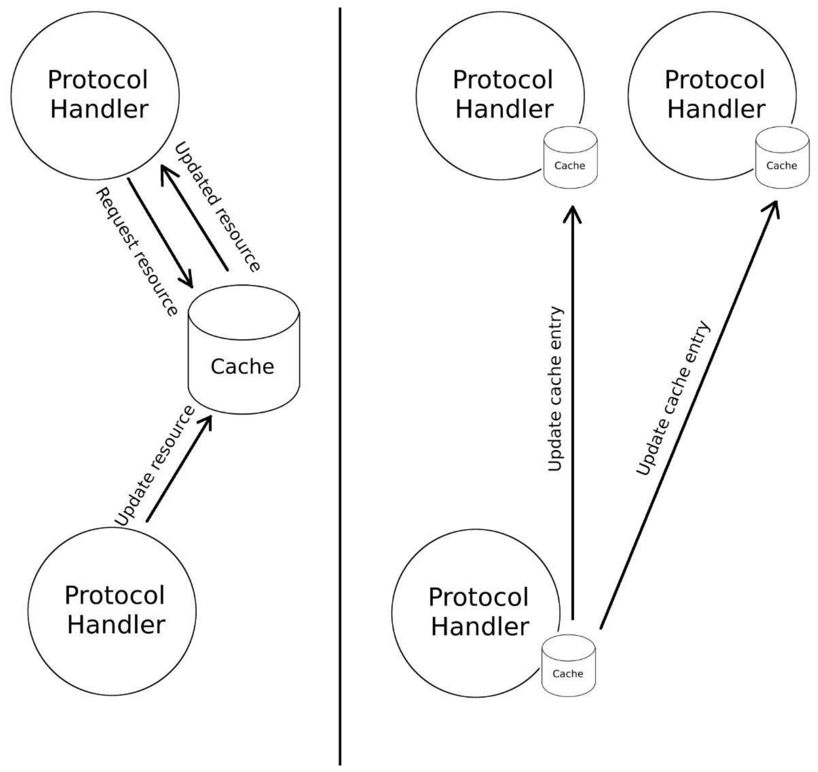


Figure 8.2: Cache use case scenario 2: per protocol optimizations in a centralized/decentralized cache

In a centralized cache:

- If a unique format for cache entries is used, update cache entries via CoAP observe doesn't introduce any more issues, provided a method to distinguish CoAP resources from the others is implemented in the system
- If entries are saved in translated form, whenever e message is received via a subscription, all the stored translated responses that correspond to the original one must be either updated or deleted.

In a decentralized cache:

- If cache entries are only stored by protocol handlers that act as clients, no further adjustments are required
- If entries are duplicated in translated form on protocol handlers that act as clients, than some sort of messaging system between caches in protocol handlers is needed to keep data consistency.

8.6 Design conclusions

In the previous sections, we presented some considerations about the design of a cache system for an HTTP/CoAP proxy. The two main designs considered are both valid and could equally fit in the proxy system. However, a decentralized cache without duplicated data is the best choice, as it has fairly small memory usage and it's easier to implement.

8.7 Implementation

The cache module has been implemented as a separate component, written in C, under the name of "Kache". As Kache have to be used to store resources in different forms (at least one per protocol), all data is handled in agnostic way.

Basic APIs

```
//initialize Kache
kache_t *kache = kache_init();

//sets the maximum number of entries stored
int kache_set_max_size(kache_t *kache, int max_size);

//sets the free function to be used on the stored data
void kache_set_freefunc(kache_t *kache, void (*k_free)(void *obj));

//kache free function
void kache_free(kache_t *kache);

//adds (or updates) an entry
int kache_set(kache_t *kache, const char *key, const void *content);

//removes an object from the cache
int kache_unset(kache_t *kache, const char *key);

//retrieves a cache entry
void *kache_get(kache_t *kache, const char *key);
```

Storage and retrieval of cache stats

To make the proxy take advantage of CoAP's observe, subscribing to popular resources, the cache must provide a simple stats system and APIs to access them. Statistics can be stored as data structures connected to cache entries. One approach consists in spawning a thread that, every arbitrary period of time, runs a procedure that iterate over cache entries and, in some way, establish which of them are popular. The supervisor routines must run in a different thread to avoid blocking for the time of execution.

Another approach is to add statistical analysis procedures directly to the cache standard functions (for example, get or set). Any time one of these functions is called, the provided procedure is executed immediately after. The best of the two approaches that, in most cases, has a lower computational cost over time.
Let's consider the "supervisor on thread" solution. Let ITIME be the time between two supervisor executions, LHISTORY the maximum length of the list of history record stored in each cache entry and LCACHE the maximum number of entries stored in the cache. Supposing the cache is full, as well as every entry history, the supervisor routine will take $LHISTORY * LCACHE$ to iterate over all histories.

In the "attached procedure" solution, every set or push function will have an overhead of LHISTORY. Let NFUNC the number of calls to the function of choice during a period of time of ITIME length, then attached routines are preferable over the supervisor thread if $NFUNC*LHISTORY < LCACHE * LHISTORY$, which is equivalent to $NFUNC < LCACHE$. As, intuitively, the number of pushes performed in a cache is less then the number of gets, this approach performs better when the routine is attached to the "set" function.

Let AVGMAXAGE be the average Max-Age of a resource (which defaults to 60 seconds in CoAP), if the statistical routine is attached to the set function, in the worst case scenario its execution costs LCACHE every AVGMAXAGE time. Using a supervisor thread that iterates over all the cache entries is convenient only in a case where $ITIME > AVGMAXAGE$ by a significant amount of time.

Furthermore, we must implement a system to avoid writes on the cache while the supervisor is reading. This adds complexity and may reduce the cache performances.

Considered the above analysis, the "attached procedure" approach have been implemented in cache, and a user can attach a statistical routine to the set function using:

```
int kache_attach_set_procedure(kache_t *kache,
                int (*procedure)(kache_entry_t *entry,void *arg),
                void *arg);
```

Optionally, a user can modify the length of the entries history (which defaults to 5);

```
int kache_set_history_length(kache_t *kache, int history_length)
```

Chapter 9

Testing environment

9.1 Devices

To test the proxy implementation, we will need some devices that can work as CoAP servers, and a border router that will help us connect to those devices. The Zolertia Z1 can serve as both.

Zolertia Z1

The Zolertia Z1 is a sensor device that comes with all that's needed to create a small CoAP demonstration application. In particular, the Z1 has the following components:

- A low power MCU (Micro Controller Unit): the Z1 is equipped with a MSP430F2617 microcontroller. This microcontroller has a 16-bit CPU with 16MHz clock speed, 8KB of RAM and 92KB of flash memory

- An 802.15.4 compliant transceiver: a CC2420 transceiver, that operates at 2.4GHz with a data rate of 250Kbps, is included on the board

- Two sensors are built-in on the board: a digital temperature sensor and an accelerometer.

- An USB interface for quick application development.

On top of these feature, the Z1 can be expanded with up to 4 external sensors like a barometer and a light sensor.

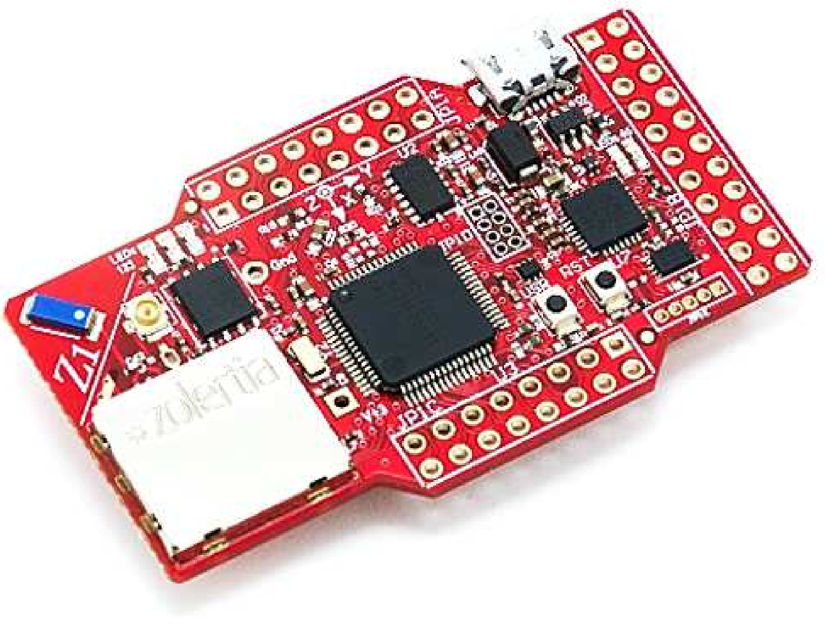


Figure 9.1: A Zolertia Z1.

9.2 Contiki

Contiki is an open source operating system especially created for the Internet of Things. Contiki was designed to work on constrained networked embedded systems and sensor networks, therefor it implies a very small footprint: typically 2 kilobytes of RAM and 40 kilobytes of ROM.

Contiki comes with a ready to use IPv6 stack that supports 6lowpan header compression, and provides a CoAP implementation. Contiki also comes with a software based power profiling mechanism to control and reduce the devices power consumption.

Finally, the Contiki operating system provides a flash-based file system called Coffee, which is optimized to store data on constrained devices at high performances.

Contiki is used in the testing setting: to convert a Z1 sensor into a border router with its RPL implementation, and to run a simple CoAP server.

9.3 The RPL routing protocol

The IoT needs a routing protocol that takes into account that networks are low power and lossy, and are formed by numerous devices. Furthermore, the routing system must take into account that the IoT traffic flows include point-to-point, point-to-multipoint and multipoint-to-point.

RPL is a routing protocol for lossy network, particularly designed for the Internet of Things. RPL specifies how to build a Destination Oriented Directed Acyclic Graph (DODAG) to control traffic flows over a constrained network. The building of the DODAG is regulated by an objective function that operates on metrics and constraints to compute the best path.

the DODAG Building Process

The DODAG building process specified by RPL starts at the root border router (indicated by the system administrator). The building process in executed through the exchange of the following kind of messages:

- DIS (DODAG Information Solicitation)
- DIO (DODAG Information Object)
- DAO (DODAG Destination Advertisement Object).

The DODAG building process proceeds with the following steps:

- The root advertises informations about the graph sending DIO messages
- The devices in the vicinity of the root receive the message and, if they received requests from other nodes, they make a decision whether to join the graph advertised or not
- A device that joins the graph becomes aware of a route that leads to the root border router, recognizing it as its parent, and computes its own rank
- If the device is itself a router, it begins to advertise the graph to its neighbors using DIO messages

Loop avoidance and detection

RPL defines mechanisms to avoid loops. The loop avoidance mechanisms are based on two rules:

- A node is not allowed to select a node as a father if that node has a greater rank. A greater rank indicates that the node is positioned in a lower position along the graph

- Nodes are not allowed to be "greedy", which means that they cannot descend further down into the graph in order to increase the number of their parents.

To identify loops, RPL specifies a method based on down/up bits. These bits indicates whether the message is moving down or up along the graph. Whenever a node has to send a message to one of its child, it indicates, using down/up bits, that the message is moving downwards. If another node receives a message and, in its routing table, finds out that the message is to be sent to a node that can be found by proceeding upwards in the graph, than it discards the message and triggers a repair routine.

Repair routines

RPL defines two kinds of repair routines: local repair and global repair. When local repair is triggered, a node tries to quickly find an alternate path when in presence of a loop. This new path may affect the optimization of the overall graph. To solve this issue, a global repair can be executed, rebuilding the entire graph.

Timeout

To minimize the control traffic, RPL defines a method called "trickle timer". This controls the interval between every DIO message multicast: less messages are sent when the graph is considered stable. The stability of a graph is evaluated using the number of inconsistencies of the network (for example, loops). When the graph is consistent, the timeout value is increased. The timeout value is decreased once any inconsistency is found on the graph.

9.4 Applications

In this section we will present the testing applications created to show some basic features of kink. For the application, we will be using two Zolertia Z1, presented section 9.1, one as a border router and one as a REST server. The Z1 that will act as a CoAP server, will have installed an accelerometer and temperature sensor. After installing the two Z1 and having a setup and running instance of kink, we can perform HTTP requests that get mapped in CoAP.

The first mapped call is on /.well-known/core, to retrieve the list of all available resources:

```
$ wget -O -  http://localhost:5683/.well-known/core 2>/dev/null

</.well-known/core>;ct=40,
</leds>;title="LED Controls";rt="Leds",
</tmp>;title="Temperature";rt="Temp",
</acc>;title="Accelerometer";rt="Acc",
</acctmp>;title="Temperature and Accelerometer";rt="TempAndAcc"
```

Every packet being sent has a maximum payload of 32 bytes, meaning that such a long message must be splitted into 6 packets using CoAP blockwise transfer, described in section 6.6. Accessible resources are the leds, that can be controlled with POST requests, the accelerometer, the temperature sensor and a combination of the last two.

A GET request to the accelerometer results in a string containing the comma separated three components of a tridimensional vector (X,Y,Z)

```
$ wget -O -  http://localhost:5683/acc 2>/dev/null
-26,31,218
```

A request to the temperature sensor returns the room temperature as a floating point value.

```
$ wget -O -  http://localhost:5683/tmp 2>/dev/null
27.6250
```

Both accelerometer and temperature sensor current values can be accessed in a single call:

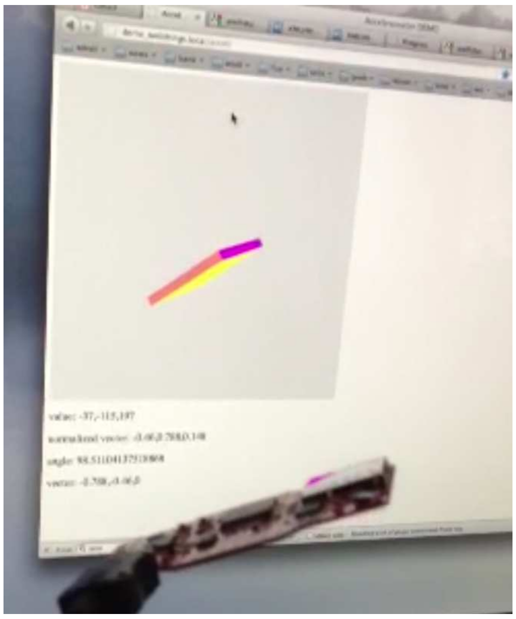
```
$ wget -O -  http://localhost:5683/acctmp 2>/dev/null
-34,36,220:26.4375
```

Accelerometer demo

To try the accelerometer call, we created a WebGL application that computes the value received as response from the sensor to show a figure that is positioned as the sensor. The GL rotation parameters are computed following these steps:

1. The string is splitted into an array of components

2. Vector components are normalized to values that range from 1 to -1
3. The resulting vector components are then swapped in order to fit the opengl axis position
4. An array is initialized, containing the components of the vector perpendicular to the figure in its initial position
5. The rotation angle is computed calculating the acosine of the dot product of the two vectors
6. The rotation vector is computed calculating the vector product of the two vectors



```
value: -37,-115,197
normalized vector: -0.46,0.788,0.148
angle: 98.5110413751908
vector: -0.788,-0.46,0
```

Figure 9.2: Accelerometer demo: the figure displayed follows the sensor's movements, shot 1

At intervals of 0.5 seconds, an Ajax request is performed to the proxy, pointing at the CoAP resource indicating the accelerometer state. Ajax calls must normally come from the same origin in order to be accepted, to avoid this issue, CORS technology [24] is used to to allow cross domain ajax requests. Static and dynamic content used for the demo, are served using Klone, an embed devices oriented web server developed by KoanLogic.

As shown in figures 9.2 and 9.2, the demonstration application was effective in representing the sensor's movements in a realistic way.

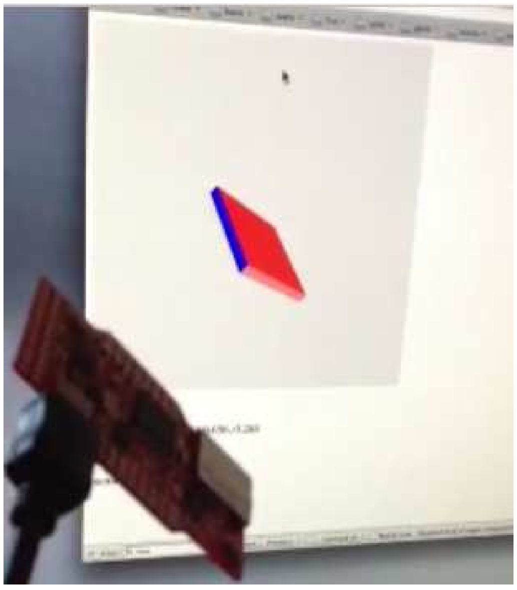


Figure 9.3: Accelerometer demo: the figure displayed follows the sensor's movements, shot 2

Chapter 10

Conclusions and future projects

10.1 Enhancing CoAP

The definition of the CoAP standard is still a work in progress, and contributors are allowed to propose new features or changes to the protocol design. We proposed a few features to be added to CoAP in order to handle communications and subscriptions between sleepy nodes.

Monitor Option

Sleepy nodes might be unable to establish a subsection on a resource hosted by another sleepy node. In fact, as they are both sleepy most of the time, the probability of an empty intersection between their respective wake periods is very high.

To allow subscriptions in such cases, we must make use of a less sleepy node that assumes the role of Proxy and acts as mediator between the two other nodes. To control such interaction, we introduce 2 options: Publish and Monitor.

The Publish CoAP option is used by a sleepy sensor to initiate a allows a sleepy sensor to use the Proxy to handle a one-way sleepy to sleepy communication. The origin server publish one of its hosted resources, by sending a PUT request it to the Proxy with a Publish option attached. If the authority transfer has succeeded, the Proxy, replies with a message with status code 2.01. Max-Age option can be use to indicate the duration of this authority transfer. If no Max-age is given, a default of 3600 seconds is be assumed.

Any client wishing to access the resource can now send the relative request through the proxy node.

The Monitor option is a variant of the Observe option that is used to regulate an observation on a resource hosted by a sleepy node using a proxy node. This is specifically useful when the requesting node has a very small duty cycle itself.

The Monitor option is used to ask a Proxy to keep a given resource fresh by observing it. The client sends a PUT request for the resource to be monitored. The request message can contain the Max-Age option indicating the duration of the subscription, otherwise this value defaults to 3600 seconds.

A monitor resource is created on the Proxy node, that from now on will maintain a fresh carbon copy of the requested resource. Multiple monitor resources, corresponding to the same target resource, can be coalesced into the same monitor object, reachable at a common URI.

On successful creation of the monitor resource, the proxy returns a 2.01 response containing the informations to reach the newly created monitor resource.

When the client, at a later time, wakes up and wants to access the monitored resource, it just requests the previously created proxy monitor resource.

In case the requested resource was not present on the origin, the Proxy will return an empty response.

A monitor object in a proxy, is deleted when all its associated resources have been de-registered or have expired. A monitor object can also be deleted in case the proxy needs to save space in memory or it went through a reboot and lost all its states. In this case, the requesting node can try to re-instantiate the monitor.

A client can also explicitly de-register a monitor by a DELETE request on the resource's registration URI.

Async Option

A sleepy node might have the necessity to communicate with another node with a small duty cycle, not to subscribe to a resource, but to perform a more complex interaction or to retrieve a resource just once.

In this case, we want the Proxy to act as a store-and-forward agent mediating the request/response exchange between two sleepy nodes. This interaction is handled this way: a node A declares the will to act onto a given resource hosted at another node B to the Proxy, and indicates the Proxy the time at which it is going to be on duty again, and willing to retrieve the response from B. The proxy will perform the request on behalf of A.

This interaction is handled using two options, Async and GetBack. As a first step, the sleepy node A asks the Proxy to act upon a resource hosted by another node B. A supplies the GetBack Option, which value indicates the time at which A thinks it will try to retrieve the response message from the proxy.

In case of success, the Proxy responds to A with a message containing an Async Option with a ticket associated to the asynchronous transaction that it is willing to handle.

The Proxy will, from now on, try to send the request to B on behalf of A, getting a response back in case of success. The Proxy will store the response until A wakes up and reissues the original request attaching an Async Option carrying the previously given transaction ticket, getting the corresponding response.

A proxy tries to immediately satisfy an incoming Async request if the requested resource is stored in its cache.

To optimize the message exchange needed by the Proxy to retrieve the requested resource from B, we introduce another option, called Sleepy, that can be piggybacked by a sleepy node on response messages to indicate: the remaining time before sleep, the expected sleep interval, and the on-duty interval.

10.2 Distribution of the code

All the code and documentation produced in order to reach this state of development, is publicly available and licensed under an open source license. The project code is hosted on GitHub [11]. The repository is called Webthings and has the following structure:

- bridge/
  - sw/
    - discopub/ – Discovery applications
    - proxy/ – HTTP-CoAP proxy code
    - lib/ – Utility libraries
      - evcoap/ – CoAP implementation based on libevent
      - kache/ – Agnostic cache module
      - urlmap/ – Regex URL mapper
  - fw/ – OpenWRT custom distribution

- docs/ – Project documentation
- wsn/ – Applications for Wireless Sensor Networks
- web/ – Web applications (like the demonstration shown in the previous chapter)

10.3 Conclusions

The completion of the previously mentioned development tasks constitutes the core logic of the KINK project, and the software framework will already be usable and customizable by third parties for their deployments. Some envisioned requirements for the next-step hardware product (phase 2) based on KINK are:

- To specialize the sys OpenWRT-based distribution for hardware with dedicated 802.15.4 support
- Integration of a routing protocol such as RPL
- Resource discovery via DNS-SD or other
- System configuration web application.

All of the above aiming at a user-friendly customizable open-black-box system, which could be turned into a commercial product for the mass. On the other hand, the field of bridging the Web and IoT networks is very wide, and the perspectives for further extension of these concepts are bright.

Bibliography

[1] About the apache http server project. http://httpd.apache.org/ABOUT_APACHE.html.

[2] An1066 - miwi wireless networking protocol stack - application notes - details. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en520606.

[3] Benchmarking libevent againts libev. http://libev.schmorp.de/bench.html.

[4] The bsd license. http://www.freebsd.org/copyright/license.html.

[5] Essential varnish modules (vmods). https://www.varnish-cache.org/project/modules.

[6] Gnu general public license, version 2 (gpl-2.0). http://www.opensource.org/licenses/GPL-2.0.

[7] Hart communication protocol and foundation. http://www.hartcomm.org/.

[8] Libebb project. http://tinyclouds.org/libebb/.

[9] Request processing in apache 2.0. http://httpd.apache.org/docs/2.0/developer/request.html.

[10] Zigbee alliance. http://www.zigbee.org/.

[11] Webthings repository. https://github.com/koanlogic/webthings, February 2012.

[12] Olaf Bergman. libcoap: C-implementation of coap. http://libcoap.sourceforge.net/.

[13] Dave Evans. The internet of things - how the next evolution of the internet is changing everything. www.cisco.com/web/about/ac79/iot/, April 2012.

[14] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext transfer protocol – http/1.1, rfc 2616. http://tools.ietf.org/html/rfc2616, June 1999.

[15] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures. http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm, 2000.

[16] K. Hartke. Observing resources in coap. http://tools.ietf.org/html/draft-ietf-core-observe-03, October 2011.

[17] Ben Laurie. Apache 2.0 hook functions. http://httpd.apache.org/docs/2.0/developer/hooks.html, August 1999.

[18] S. Loreto, P. Saint-Andre, S. Salsano, and G. Wilkins. Known issues and best practices for the use of long polling and streaming in bidirectional http. http://tools.ietf.org/html/rfc6202, April 2011.

[19] T. Minuzzo, A. Bergamasco, S.Carniel, and M. Sclavo. Un data server per l'accesso unificato ai dati scientifici, nell'ottica di un sistema integrato per le osservazioni dell'ambiente marino. http://www.garr.it/eventiGARR/conf10/docs/minuzzo-abs-conf10.pdf.

[20] G. Montenegro and C. Schumacher. Ipv6 over low-power wireless personal area networks (6lowpans): Overview, assumptions, problem statement, and goals. http://www.microchip.com/stellent/idcplg?IdcService=SS_GET_PAGE&nodeId=1824&appnote=en520606, August 2007.

[21] A. Rocchi, S. Pierattini, G. Bracco, S. Migliori, F. Beone, A. Santoro, and C. Sciò. Faro: Accesso web a risorse remote per l'industria e la ricerca. http://www.garr.it/eventiGARR/conf10/docs/rocchi-abs-conf10.pdf.

[22] Z. Shelby. Core link format. http://tools.ietf.org/html/draft-ietf-core-link-format-10, January 2012.

[23] Z. Shelby, K. Hartke, C. Bormann, and B. Frank. Constrained application protocol (coap). http://tools.ietf.org/html/draft-ietf-core-coap-08, November 2011.

[24] Anne van Kesteren. Cross-origin resource sharing. `http://www.w3.org/TR/cors/`, July 2010.