Observing CoAP groups efficiently

Isam Ishaq[a,b], Jeroen Hoebeke[a], Ingrid Moerman[a], Piet Demeester[a]

[a] *Ghent University - iMinds, Department of Information Technology (INTEC)*
Gaston Crommenlaan 8 (Bus 201), B-9050 Ghent, Belgium
{isam.ishaq,jeroen.hoebeke,ingrid.moerman,piet demeester}@intec.ugent.be
Tel: +32 (0)9 33 14900
[b] *Said Khoury IT Center of Excellence (SKITCE)*
Al-Quds University, Abu Deis, Jerusalem 51000, Palestine
isam@alquds.edu

Abstract

It is envisioned that by the year 2020 the Internet will contain more than 50 billion devices, among which the majority of them will have constraints in terms of memory, processing power or energy. As a consequence, they are often unable to run current standard Internet protocols, requiring special, optimized protocols. A number of these protocols, covering the different layers of the protocol stack, have been developed and standardized lately. At the application level, the Constrained Application Protocol (CoAP) is proposed by the IETF as an HTTP replacement that is suitable for constrained devices. CoAP is a very light-weight base protocol that can be extended with optional specifications to satisfy specific use case needs. Two important optional specifications are *observe*, allowing monitoring of a CoAP resource over a period of time, and *group communication*, supporting interactions with multiple CoAP devices at once. Currently, these two optional specifications do not work together, i.e., it is not possible to gain the benefits of both of them at the same time. In this paper we present an alternative and novel approach to CoAP group communication that works well with the CoAP observe extension. In addition, it enables to perform operations on the observed results, bringing intelligence closer to the data sources.

Keywords: Internet of Things, CoAP, sensors, wireless sensor networks, group communication, CoAP observe, entities

1. Introduction

The *Internet of Things* (IoT) is continuously expanding in many directions. It is expanding horizontally as new application domains start to rely on the opportunities offered by the Internet. At the same time, it is expanding vertically as within established connected domains, the types and numbers of devices connected to the Internet rapidly increase. Many of the newly connected things are connected using embedded devices that are typically optimized for low-cost and low-power consumption. These devices are constrained in their resources (CPU, RAM, ROM ...) and thus unable to run standard Internet protocols, which were originally designed with certain expectations of the devices' resources in mind. The local networks that connect these constrained devices together are often referred to as *low power and lossy networks* (LLNs).

In the last few years, many efforts have been put into the extension of standardized Internet technologies to constrained devices. Meanwhile, open standards exist to cover the complete networking stack and thus enabling, among many other things, compatibility between devices of various vendors [1].

Most noteworthy standardization efforts targeting the extension of Internet protocols towards constrained devices are the efforts of the *Internet Engineering Task Force* (IETF). Initial IETF efforts focused on the networking layers and resulted in the integration of constrained devices and LLNs in the IPv6 Internet. Later, the IETF started targeting the efficient integration of constrained devices in web services and therefore established the *Constrained RESTful Environments* (CoRE) working group. CoRE aims to allow access to constrained devices in a RESTful way, similar to how most information on today's Internet is accessed over HTTP. CoRE takes as protocol design requirements the communication needs of *machine-to-machine* (M2M) applications such as smart energy and building automation [2].

So far, the main achievement of CoRE is the standardization of the *Constrained Application Protocol* (CoAP), which the IETF sees as an embedded counterpart of HTTP [3]. Constrained devices are turned into embedded web servers that make their resources accessible to Internet clients via the CoAP protocol. Typically, each of the constrained servers has at least one CoAP resource that may be queried by clients to obtain information about the devices themselves (e.g., battery level), about the environment that they monitor (e.g., temperature of the room), or to trigger the devices to perform real-world

actions (switch the light on). These CoAP resources are identified by a *Uniform Resource Identifier* (URI) such as `coap://[aaaa::1]/temperature`.

As various IoT applications may have different communication requirements and the devices may be very heterogeneous in their capabilities, CoRE designed CoAP in way that the base protocol is kept as efficient and as simple as possible so it would run on even the most constrained devices. Features that might not be needed by all devices are standardized in a set of optional separate specifications. Two of the most important CoAP optional specifications are:

Group Communication for CoAP. Section 8 of the base CoAP specification targets the well-recognized need for addressing several CoAP resources as a group, instead of addressing each resource individually. For example, this might be used to turn on all the CoAP-enabled lights in a room with a single CoAP request triggered by toggling the light switch. CoAP group communication relies on IP multicast to deliver the CoAP request to all group members. RFC 7390 [4] is an experimental RFC that provides best practices to use CoAP multicast messages and defines an optional interface for group management.

Observing Resources in CoAP [7], is currently an Internet Draft that allows clients to register their interest in receiving notifications from CoAP servers once there are changes to the values of the observed resource. By doing so the client does not need to continuously pull the resource to find out whether it has changed its value. This typically leads to more efficient communication patterns that preserve valuable device and LLN resources. For example, this optional CoAP extension might be used to monitor the temperature inside a refrigerated container.

Unfortunately, these two important optional CoAP specifications have not been designed to work together, i.e., it is not possible to combine these two specifications to observe a group of resources, e.g., to observe the status of the room lights group. This means that if a client needs to always have an up-to-date status of a group, it would need to either continuously pull the group or observe the members of the group individually. In other words, the client loses either the benefits gained by observing or the benefits gained by grouping. Both cases are suboptimal. In the former case, a lot of unnecessary traffic will be generated. In the latter case, clients would need to have the

knowledge about the individual group members and to maintain the observe relationship with all of them individually. Either way leads to more complex and less flexible client applications.

Generally, there are two main approaches to solve the problem of observing CoAP groups of resources. In the first approach, multicast is used for group communication and the observe draft will need to be adapted to be able to deal with it. The use of multicast has certain benefits such as reducing the number of packets needed to deliver the request to the members. However, multicast also has its limitations such as poor reliability and being cache-unfriendly. In the second approach, multicast communication is avoided and other means for group communication are used and thus it becomes possible to observe resources in accordance with the observe draft. In this paper we examine the second approach by building up on our previous work [5], in which we have proposed an alternative method for CoAP group communication and submitted it as an Internet Draft [6]. Our method uses an *Entity Manager* (EM) acting as an aggregation point for the groups and offering CoAP resources that allow clients to create and interact with CoAP groups using unicast messages. In order to provide an alternative way for observing groups of resources, we present in this paper a major extension to our group communication solution [5], namely the efficient support for CoAP observe at the level of a group.

In this paper, we also propose to extend the *Entity Operations* concept, which we briefly introduced in [5], in order to support operations on the values of observed resources. By using Entity Operations it becomes possible to further reduce the communication needs between the group and the observing client(s). This efficient way of observing a group of resources is a way of providing distributed intelligence and is in line with current trends in research and industry to bring processing closer to the data sources [8, 9]. Our approach to observing a group of resources is also in the spirit of CoAP, i.e., processing of data and exposure as web services.

In summary, we present the following novel contributions in this paper:

- The observation of groups of CoAP resources, carefully managed by an Entity Manager, thereby pushing changes in a group or changes in processed information to the observing client(s).

- The possibility of applying operations to the observed resources within a group (Entity Operations), further reducing the amount of communication.

- An improved architecture that enables to easily plug in support for new Entity Operations.
- Fine-grained control over the precision and frequency of notifications a client wants to receive about a group by assigning properties to the Entities (optimization techniques)
- An analysis and evaluation of the proposed optimization techniques.

The remainder of this paper is organized as follows. Section 2 explores the related work on CoAP group communication and the observation of groups. CoAP and a few needed optional specifications are explained in Section 3. In Section 4 we first summarize our previously introduced solution and then describe in detail how we extended it to support observation of groups. Next, in Section 5, we present a few optimizations to reduce the communication overhead of observing CoAP groups. In Section 6 we present our implementation and evaluate the functionality and the performance of our solution. Finally, Section 7 concludes this work with a summary and outlook.

2. Related work

As mentioned in the introduction, the basis of our work are the standardization efforts of the IETF CoRE Working Group, i.e., the Constrained Application Protocol (CoAP) [3] and two of its most important optional specifications: Observing Resources [7] and Group Communication [4].

The first optional specification specifies an observing mechanism that allows avoiding the continuous polling of CoAP resources by letting clients register for receiving notifications upon changes in the resources' values. Although this optional specification was still an Internet Draft at the time of writing this article, this draft has undergone many revisions by the Working Group and is expected to be approved without major changes.

The second of those two optional specifications was developed to address the group communication needs in CoAP and has been published as the experimental RFC 7390. This RFC specifies how CoAP should be used in a group communication context. It provides an approach for using CoAP on top of non-reliable IP multicast. Certainly, the use of multicasts allows reducing the amount of requests in the LLN, by sending one request to several destinations at the same time. However, this method for CoAP group communication is not supported with CoAP observe, since the latter only allows

unicast messages. In addition, the use of multicast has other limitations such as being not cache-friendly and not supporting secure communication (see Section 3.3).

An earlier draft version of Group Communication for CoAP was the basis for the work presented in [10], in which web services based CoAP multicasts were used to access data from *Building Automation Systems* (BAS). It shows how using multicasts allows creating basic building control scenarios without the need of a central control unit. Certainly this approach has several advantages such as eliminating the need for a control unit, often a lower power consumption than using unicasts and its suitability in many non-critical use cases (due to the lack of reliability of multicasts). However, since this approach is based on IP multicast, it does not support CoAP Observe and exhibits the other limitations of multicasts as discussed in Section 3.3.

The authors of [11] present a lightweight multicast forwarding solution specifically designed for the requirements of service discovery in LLNs. Since in such cases group management is not required, the authors designed a simple alternative to multicast by relying on filtered flooding techniques to broadcast messages to all nodes in the local network while avoiding the creation of forwarding loops. The authors argue that broadcasting in ContikiMAC has certain drawbacks such as higher than necessary number of transmitted packets and lack of reliability. In order to tackle these issues, the authors replaced local broadcast with multiple unicast transmissions. By doing so, the number of transmitted packets, the packet propagation delay in the network and the reliability were improved at the cost of a slightly increased footprint. This approach is similar to ours in the way it relies on unicast rather than multicast to achieve group communication. However, unlike ours, this approach is suitable only for the cases where messages need to be sent to all nodes in the network.

In a machine-to-machine context, group communication is often more than just the communication aspect. If a machine receives data from multiple sources, the machine needs to know more details about the data it is getting from the various sources in order to be able to process this data correctly. This includes the need for strict typing of data to be able to understand the contents of the resources. Several initiatives that provide guidelines and standards in this regard exist. For example, the *CoRE link format* [12] provides a way to specify a *Resource Type* by using the `rt` attribute. In our work we rely on those standards to figure out how to combine different resources together. Combining notifications and performing operations on

them before notifying the observer brings data processing closer to the data sources. Lately, several works in the industry and research community focus on distributed intelligence in the IoT field and in bringing data processing closer to the data sources (Fog computing). A major driver for these works is that connectivity to the Cloud is not infinite and not always cheap [15, 8, 9]. In that respect, our work can be seen as an innovative application enabler that can be supported by the Fog, which is one of the research areas identified in [15].

An approach somewhat more similar to ours, also using the notion of an entity and supporting CoAP Observe, has been presented in [16]. The aim here is to annotate real-world objects by using entities that are automatically created based on semantic information, which resides on the constrained devices. One problem of using semantics on constrained devices is that semantics are verbose and can easily require a lot of memory. This makes them difficult to fit on constrained devices without applying compression techniques [17]. Further, in our approach users can create entities as required and we address important aspects related to entity validation and behavior.

The authors of [18] present an extension to CoAP called SeaHttp that enables communication with a group of resources. Similar to our work, SeaHttp also uses unicasts to realize group communication. The authors propose to extend CoAP with two additional methods (BRANCH and COMBINE) to allow members to join and leave groups without the need for a separate group manager. This means that members should have the intelligence to know which group they should join/leave. Constrained devices will not have this intelligence, so similar to our solution, we believe that SeaHttp does need a "manager" to inform the devices so they can take appropriate actions. Furthermore, BRANCH and COMBINE can maybe reduce the number of messages; however, the trade-off is the need to implement a new mechanism. It is better to use an approach that can be plugged in into any existing network without major modifications (or at least not a modification to every node). The article does not discuss whether SeaHttp resources support CoAP Observe or if the use of caches will still be possible with them. However, should this be possible then also the caches should be extended accordingly. Finally, this approach does not have the flexibility we target, since group members have to be reprogrammed with the groups they should join each time the requirements of the user changes.

The authors of [19] propose a scalable in-network storage solution for

sensor networks benefiting from the available memory of the smart objects. This solution is built on agents and can autonomously manage the system without humans in the loop. To achieve this autonomous management, the solution relies on CoAP multicast messages to build and manage hierarchical zones that follow the structure of the building in which the system resides. The authors propose to combine the observe and multicast group communication in order to be able to update more than one storage location at once, without the need for those to register with the resource individually. This is achieved by sending the notification to the multicast address of the storage. This solution demonstrates another use case that needs to combine observe with group communication, but takes a different approach than ours to achieve it.

To our knowledge, these are the only works that explore communication solutions for interacting with a group of CoAP-enabled constrained devices. Next to these, there exist other solutions to realize group-like communication in constrained environments without using CoAP. For example, the Message Queue Telemetry Transport (MQTT) protocol is another application layer protocol designed for constrained devices [20]. MQTT uses a topic-based publish-subscribe architecture, i.e., clients utilize the services of a *Broker* to subscribe to *Topics* and get all the *Messages* that are published to that topic. Unlike CoAP, MQTT relies on TCP as underlying transport protocol and thus inherits its reliability. While the base CoAP does not provide any *Quality of Service* (QoS), MQTT provides its own QoS mechanism. MQTT provides its own way of group communication with observe support, by allowing multiple publishers to publish to the same topic and by allowing multiple clients to subscribe to it. This can be seen as a form of group communication which exhibits some similarities with our proposed approach. However, it does not adhere to the REST principles that are commonly used in the Internet and, additionally, it does not provide the possibility to aggregate and manipulate notifications that are sent to the clients.

3. CoAP Overview

The focus of this paper is to enable observing a group of resources from a service/application perspective in a way that is in line with existing and ongoing standardization activities in the field of IoT. In the last few years a lot of effort has been put in defining a standard application protocol, similar to HTTP, but more suitable for constrained devices, namely CoAP. The base

CoAP protocol is defined in RFC 7252 [3] in conjunction with a number of additional specifications. In this section we briefly introduce the base CoAP specification and those optional specifications that are relevant to our group observing work.

3.1. Base CoAP - RFC 7252

The main idea behind designing CoAP was to create a protocol that uses the same RESTful model as HTTP, but it is much lighter so that it can run on constrained devices [21, 22]. The *Representational State Transfer* (REST) architecture uses a *request/response model* in which clients exchange representations of resources with servers. A client interested in the state of a resource initiates a *request* to the server, which then returns a *response* with a representation of the resource that was current at the time of the request. A *resource representation* either captures the current or the intended state of the resource.

In order to reduce the resource requirements of CoAP, it was designed to have a much shorter header and lower parsing complexity than HTTP. CoAP uses a compact (4-bytes) base binary header that may be followed by optional extensions. The CoAP interaction model is similar to the client/server model of HTTP. A client can send a CoAP request, requesting an action specified by a method code (GET, PUT, POST or DELETE) on a resource (identified by a URI) on a server. The CoAP server processes the request and sends back a response containing a response code and an optional payload.

A message that does not require reliable transmission can be sent as a Non-confirmable Message (NON). This type of messages is not acknowledged and thus might get lost without the client and the server noticing it. Unlike HTTP, CoAP deals with these interchanges asynchronously over a datagram-oriented transport layer such as UDP and thus supports multicast requests. This allows the use of CoAP for point-to-multipoint interactions, which are commonly required in automation. Multicast CoAP requests are sent using NONs.

Since UDP does not provide reliable communication, optional reliability is supported within CoAP itself. This is done by using Confirmable Messages (CONs) to implement simple stop-and-wait retransmissions with exponential back-off.

To be able to offer communication needs that cannot be satisfied by the field of the base binary header alone, this base header may be followed by one or more of the following optional fields: Token (to correlate requests and

responses), Options, and Payload. As an example of a simple CoAP option consider the `Max-Age` option. This option indicates the maximum time a response may be cached before it is considered not fresh. If this option is not included in any CoAP response, the client (or cache) should assume that the response will be fresh for 60 s and thus will not query it again within this period. Another example is the `Uri-Query` option, which is a string that specifies one argument parameterizing the resource. Clients can use this option for instance to request that the server should process the response in a different way.

In machine-to-machine (M2M) applications where there are no humans in the loop, it is important to provide a way to discover resources offered by constrained servers. For HTTP Web Servers, the discovery of resources is typically called Web Linking - RFC 5988 [23]. The use of Web Linking for the description and discovery of resources hosted by constrained web servers (CoAP or HTTP) is specified by the CoRE Link Format - RFC 6690 [12]. This RFC defines a well-known relative URI `/.well-known/core` as a default entry-point for requesting a list of links to resources hosted by a CoAP server. Once the list of available resources is obtained from the server, the client can send further requests to obtain the value of a certain resource. The example in Figure 1 shows a CoAP client requesting the list of the available resources on a CoAP server (`GET /.well-known/core`). The returned list (in CoRE Link Format) shows that the server has, amongst others, a sensor resource called /s/t that, when queried, returns the temperature in degrees Celsius. The client then requests the value of this resource (`GET /s/t`) and receives a plain text reply from the server with the value of the current temperature as payload of the message (`23.5`).

3.2. Observing Resources

CoAP follows the REST model, i.e., clients initiate requests to servers in order to exchange representations of resources (see Section 3.1). A response returned by a server is a representation of the resource at the time of the respective request. The REST model does not work well when a client is interested in having a current representation of a resource over a time period. A few approaches exist for HTTP to solve this issue, e.g., repeated polling or HTTP long polling [24], However, these approaches generate significant complexity and/or overhead and thus are less applicable in constrained environments. In this subsection we describe the *Observing Resources in CoAP* optional extension that allows clients to register with servers their interest

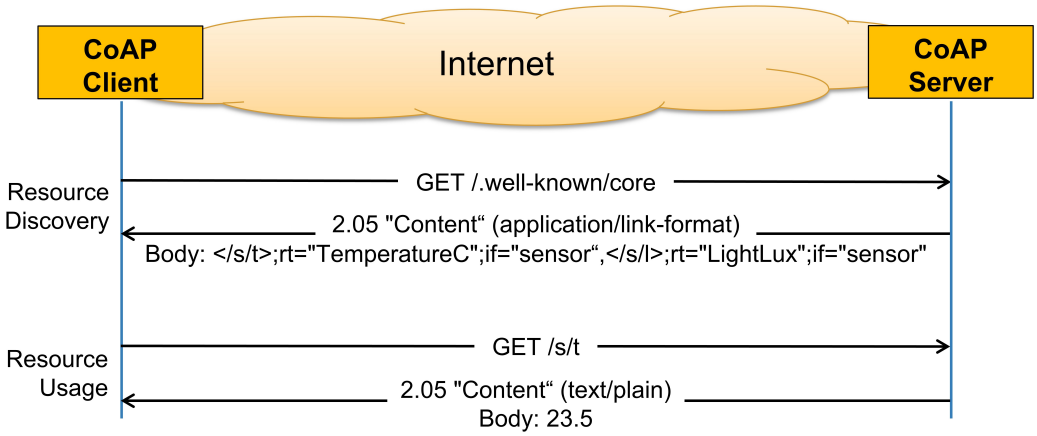


Figure 1: An example of Constrained RESTful Environments (CoRE) direct resource discovery and a Constrained Application Protocol (CoAP) request.

to receive notifications whenever a resource representation is changed. This extension keeps the architectural properties of REST but avoids the repeated pulling of resources.

3.2.1. Observe Option

The *Observing Resources in CoAP* extension defines a new CoAP option and an associated protocol for its use. Thus, it extends the base CoAP protocol with a mechanism for a CoAP client to *observe* a resource on a CoAP server. When a client is interested in observing a resource over a time period, it adds the `Observe` Option to its request to the server. The server replies to the client with the current representation of the resource and updates the client about any changes to the representation as long as the client is interested in the resource.

Figure 2 shows an example of a CoAP client *registering* its interest in a resource and receiving two *notifications*: the first with the current state upon registration, and the second upon a change to the resource state. Both the registration request and the notification are identified as such by the presence of the `Observe` Option. The client sets the value of the `Observe` Option to zero to indicate a registration request. In notifications, the server sets the value of the `Observe` Option to provide a sequence number that the client can use for reordering notifications. This number does not have to be sequential, but must be higher than the previous notifications. In order to make it possible for the client to correlate notifications with requests, the

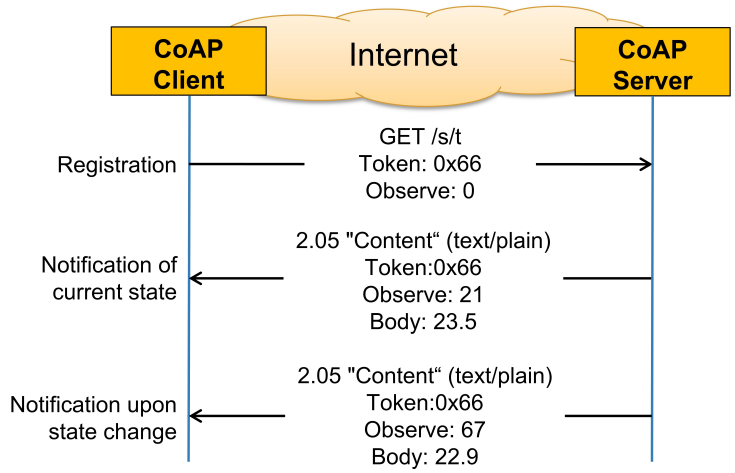


Figure 2: Observing a resource in CoAP. Clients register their interest in observing a resource. Servers then notify clients whenever the resource changes its value. The Token must be the same in order to match the notifications to the registration.

server must add the same token that the client used in the registration to all notifications.

3.2.2. Observe Consistency Model

The goal of the Observe protocol is to keep the resource state observed by the client as closely in sync with the actual state at the server as possible. However, due to signal propagation delay and the lossy nature of LLNs, it cannot be avoided that the client and the server become out of sync sometimes. Also, due to CoAP's congestion control mechanisms and the lossy nature of LLNs, clients cannot rely on observing every single state that a resource might go through.

While the Observe protocol can not guarantee that clients will always have an up-to-date representation of the observed resource, it follows a best-effort approach to get to this goal as close as possible. The protocol uses the `MAX_AGE` option (Section 3.1) to label notifications with a maximum duration up to which it is acceptable for the observed state and the actual state to be out of sync. The protocol is designed on the principle of eventual consistency. This means that, if the resource does not undergo a new change in state, eventually all observing clients will have a current representation of the latest resource state.

3.2.3. Observe Extension to Web Linking

CoAP servers can indicate to the clients that a resource is observable using Web Linking. To do so, the server extends the resource's entry in `/.well-known/core` with a new attribute: `obs`. When this attribute is present (it should not contain a value) it is a hint that the resource is suitable for observation. For Example, to hint that the resource is observable, the entry for `/s/t` from Figure 1 will become:

```
</s/t>;obs;rt="TemperatureC";if="sensor"
```

3.3. Group Communication

The IETF CoRE working group has recognized the need to support a non-reliable multicast message to be sent to a group of devices to manipulate a resource on all the devices in the group. Therefore, they have included group communication in the base CoAP protocol and have developed the "Group Communication for CoAP - RFC 7390" [4], which provides guidance for how the CoAP protocol should be used in a group communication context. Group Communication refers to sending a single CoAP message to all members of a specific group by utilizing UDP/IP multicast for the requests, and unicast UDP/IP for the responses (if any). This implies that all the group members (the destination nodes) receive the exact same message.

The use of multicast is efficient in sending the requests, but does not affect the number of responses sent by the members since they are sent as unicasts. However, the use of multicasts has its limitations and challenges:

- The most prominent limitation is the **lack of reliability**, which makes it not suitable for all use cases.
- Another important limitation is the **cache-unfriendly** nature of multicasts preventing possible reduction of requests and replies by utilizing caches. Depending on the use case and network topology, the reduction of packets as a result of using a cache can be better than the reduction obtained from using multicasts.
- Also, multicasts are **not useful when a single user action needs to trigger different sensor requests**, since one multicast request delivers the same message to all group members.
- **Secure communication with the group members is not possible**, since all communication based on this RFC operates in CoAP NoSec (No Security) mode.

- **Multicast is not supported on all LLN MAC protocols**, especially MAC protocols that use *Radio Duty Cycles* (RDC) to shut down their radios when not in use. For example, Xmac does not support multicast since it shuts down its receiver to avoid overhearing [25].
- Finally, and for our current work most important limitation, is that the **CoAP Observe Option does not support multicast** transport mode.

4. Flexible and Observable Entities

In [5] we have introduced our flexible unicast-based group communication solution for CoAP-Enabled devices. In the following subsections we first summarize our previously introduced solution and present its improved architecture (Section 4.1) and then show how we extended it to support observation of groups (Section 4.2).

4.1. Group Communication Using Unicasts

Our aim was to create an intermediate level of aggregation to be able to easily manipulate a group of resources across multiple smart objects. To avoid increasing the footprint of the constrained devices, we used the same technology as used to manipulate individual resources, i.e., CoAP, and extended it accordingly. Such a group of resources are called an *entity* and the resources themselves are called the entity *members*. An entity can be created, used or manipulated through a single CoAP request. When a client requests the creation of an entity, the EM performs a complete *validation* of the entity.

Furthermore, we have introduced in [5] the notion of profiles for the created entities. The use of entity profiles allows the client to specify in more detail how the entity should behave (e.g., if it should use confirmable or non-confirmable CoAP messages), and, through updating the profile, allows manipulation of this behavior. As such, we combined ease of creation, ease of usage and flexibility in behavior into a complete solution for interacting with CoAP resources from different objects inside a LLN. By building upon standardized concepts, the impact on the constrained devices was limited.

4.1.1. System Overview

We call the component that manages the entities, the *Entity Manager* (EM). This component, which can reside, e.g., on the Border Gateway of the

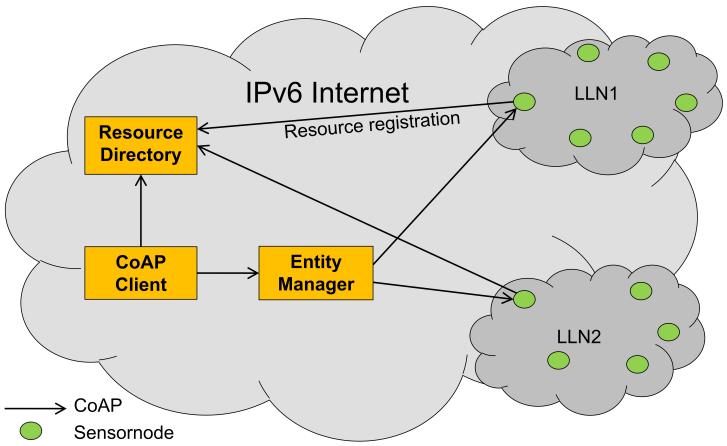


Figure 3: Clients create entities consisting of several smart object resources on the Entity Manager. Clients can optionally query a resource directory to discover the existence of the resources.

LLN, is responsible for maintaining entities that are created from groups of resources residing on CoAP servers (i.e., sensors and actuators) inside the LLN. Clients on the Internet can interact with an EM to create new entities and/or customize how these entities should behave. Optionally clients can elect to contact a resource directory [26] in order to discover which resources are available in the network. Figure 3 shows an overview of the involved components.

The EM functionality does not have to reside on a dedicated device. Theoretically, any CoAP server can be extended to become an EM (Figure 4). The choice of the most appropriate location to put the EM functionality depends on the size and topology of the network. For example, it can reside on a smart object in the constrained network with enough resources, in the Cloud, on the client device itself, or on a gateway at the edge of the LLN. The latter case has the added benefit that security can be centrally managed besides offloading the processing from constrained devices. One can also decide to implement multiple EMs (at the same or at different locations) to avoid having a single point of failure and thus improving reliability, availability and scalability.

Regardless of the location of the EM, it will serve as a proxy between the client and the constrained devices. Client requests will be sent to the EM, which will analyze and verify the requests and then issue the appropriate

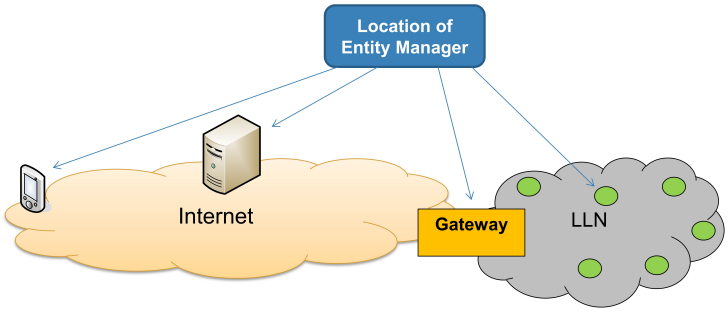


Figure 4: The Entity Manager (EM) functionality can be integrated into any CoAP server. The optimal location for the EM depends on the use case.

requests to the constrained devices using CoAP. Once the EM receives responses from the constrained devices, it will combine them according to the needs of the client and will send back an aggregated response to the client.

When a client tries to create a new entity consisting of a group of resources inside LLNs, the EM performs a sanity check on the request in order to make sure that the resulting entity would make sense. We call this sanity check the *Entity Validation*. For example, it verifies that the resources inside the entity are valid, whether they support a certain content format and whether their data can be aggregated. Customization of the entity behavior is accomplished by creating *profiles* for the entities. A profile of an entity can specify for example whether to return the values of all resources in the entity, only the computed average of all values or a subset of all values. Figure 5 shows a high-level structure of the Entity Manager. It shows that the EM contains three databases:

Entity Database: In this database, all entities are stored along with their profiles as defined by the user.

Clients Database: In this database, the EM stores the details of the clients that it is currently serving. These details include any customizations of the entity properties for the particular client and whether this client is observing the entity. If the client is no longer observing the entity, the EM removes the client details from the database once the EM sends a reply to the client. However, if the client is observing the entity, the EM keeps the client details until it stops observing.

Capabilities Database: This optional database provides rules and knowl-

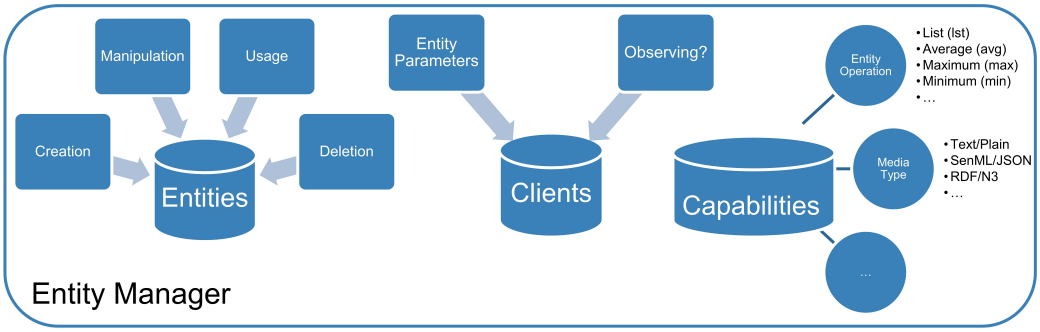


Figure 5: The EM manages the live-cycle of entities from creation until deletion and manages the relationship to the clients that are using the entities. In addition, it contains a knowledge database about the capabilities it can use to match user requests with sensor capabilities.

edge that the EM uses to match user requests with sensor capabilities. This can be as simple as translating a request for temperature in degrees Celsius while obtaining the data from a sensor that only supports Fahrenheit. It can also be more complex, e.g., converting resource representations from one content format into the other. It also provides a library of operations that can be applied on the individual member responses in order to create the aggregated response sent to the client.

4.1.2. Entity Creation

To facilitate the creation and manipulation of entities, the Entity Manager (EM) offers a CoAP resource "/e". We call this resource the Entity Management Resource. This interface only supports the CoAP POST request method. As payload of the request, it expects a collection of resources in CoRE link format [12], which together should form the entity. In the response, the `Location-Path` CoAP option is used to specify the name of the newly created resource. In the current design, the payload of the response is in plain text and describes the results of the validation tests performed by the EM on the collection of resources.

Thus, when a client wants to create an entity consisting of several members, it has to compose a CoAP POST request and send it to the Entity Management resource on the EM. The EM creates the entity, assigns it a unique URI, and stores the entity in the entity database for future usage. Then the EM starts the entity validation process (see [5] for details). The client is informed about the URI to use in order to access or further customize

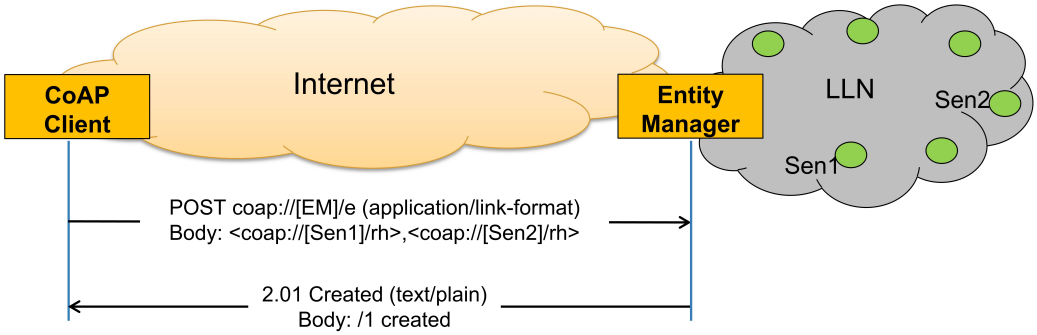


Figure 6: A CoAP client requesting from an EM to create a new entity that contains two resources.

the newly created entity and about the results of the validation of the entity.

An example of the entity creation process is shown in Figure 6. In this simple example the client requests the creation of an entity consisting of two members: `coap://[Sen1]/rh` and `coap://[Sen2]/rh` , with `Sen1` and `Sen2` the IPv6 addresses of the two sensors. The EM creates the new entity, assigns it the URI `/1` and informs the client about the newly created entity. From now on, any client can access the newly created entity by accessing the `/1` resource on the EM. Please note the validation process is not shown in Figure 6 for simplicity.

At creation time, the client can use optional URI-Query CoAP options with the POST request to specify the name of the entity to be created or to customize the default behavior of the entity. For example, to create the entity "room_humidity" that returns by default the minimum value of all members when queried, a POST to `coap://[EM]/e?path="/room_humidity"&eo="min"` is needed. We will discuss customization of the entity behavior in more detail in Section 5.

4.1.3. Entity Usage

When a client wants to use an existing entity, it sends a standard CoAP request to the entity resource on the EM. This request may include optional Uri-Queries to customize the behavior of the entity for this particular request. Simplified, i.e., ignoring all error conditions, the EM handles entity usage requests as shown in Figure 7. First, the EM analyses the request in order to obtain the set of individual requests it needs to send to the entity members. For this, information residing in the Clients Database and Entities Database

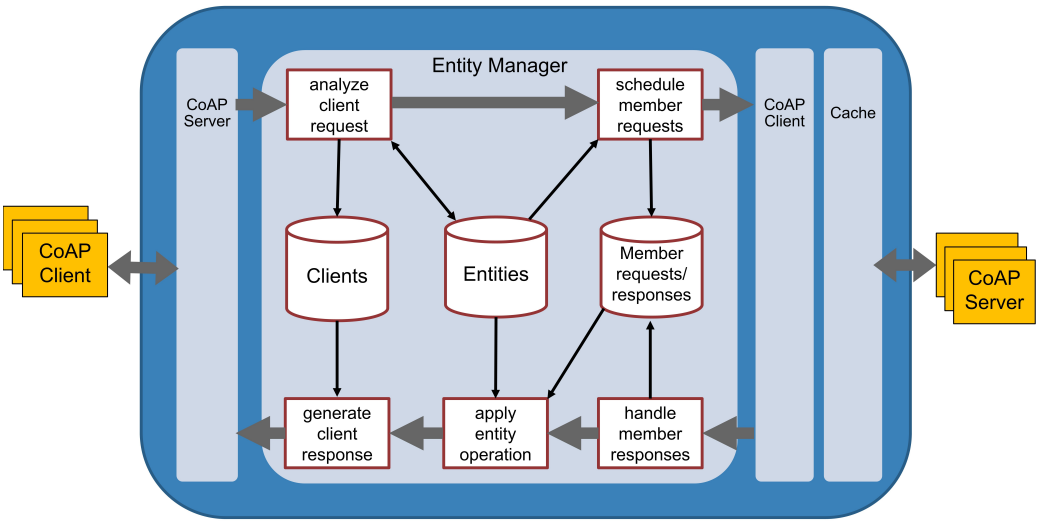


Figure 7: Handling client requests to use existing entities.

is being used. Next, it creates the required requests and schedules them for delivery to the members. Each request passes then through a cache, which decides whether to reply directly (if the cache has a fresh reply) or to actually pass the request to the member. Either way, the EM eventually receives a response for its request and stores it in its temporary responses storage. Once the EM has received all required responses, it uses its capabilities database in order to convert all replies to a common format and then applies the needed Entity Operation to obtain a single response and send it to the client.

Figure 8 shows an example of using the entity that was created previously in Figure 6. The client issues a GET request on the entity's resource `/1`. This results in the EM issuing two GET requests to the individual members, waiting for replies from both of them and then sending both results in one combined response back to the client. In this case the reply from the EM to the client was in senml+json format [27]. This format is considered simple enough to be constructed by constrained devices, and at the same time to be parsed efficiently by servers.

The client can decide to query the entity using its default behavior as described in the entity profile or to customize its behavior. To customize the behavior the client can include URI queries in its request to the entity. An example of the supported URI queries that can currently be used is the Entity Operation (eo). For example, the client should use the URI:

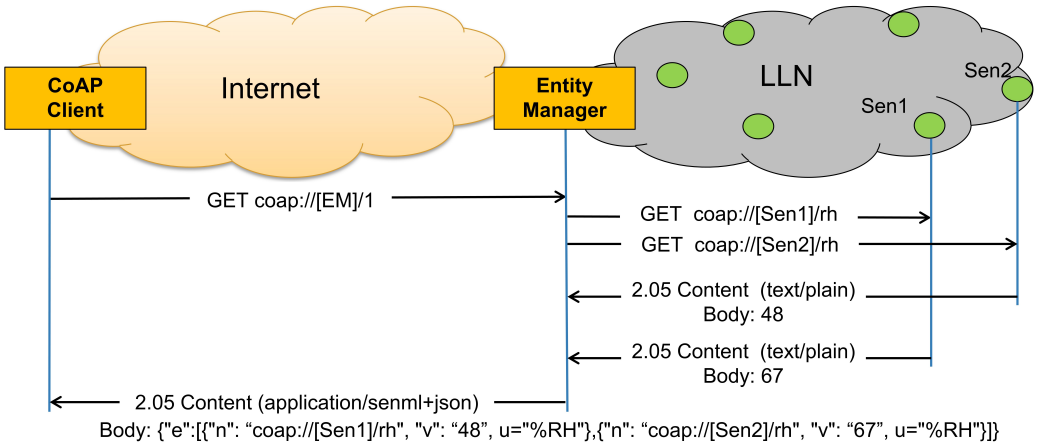


Figure 8: A CoAP client requesting from an EM to obtain the values for the entity that was previously created in Figure 6.

`coap://[EM]/1?eo="avg"` to obtain the average value of the two relative humidity sensors of the entity `/1` in Figure 8.

4.2. Group Observation

When a client creates an entity, it can indicate to the EM that this entity should be observable by adding a Uri-Query `obs=1` to the entity creation request. In such cases, the EM extends the entity validation steps to check that all the entity members are indeed observable. Typically, this information is contained in the resource's entry in `/.well-known/core` (see Section 3.2.3). If all members are observable the entity is also flagged as observable by adding the `obs` attribute to the entity resource in `/.well-known/core`.

When a client is interested in observing the group, it establishes an observe relationship with the entity resource in the same way it would start an observe relationship with any other observable resource, i.e., by sending a GET request with the `Observe` Option set with a value of zero. The EM adds the client to its *list of observers* and in turn starts observing all the members of the group in the same manner.

When an observed group member changes its value, it sends a notification to the EM informing it about the new value. The EM manager stores the new value and potentially notifies the client about the new change either immediately or after a certain time. In the remainder of this article, we call the notifications from entity members to the EM *member notifications* and

call the notification sent from the EM to the clients *client notifications*. The decision whether to notify the client and when to do so is based on three configurable entity properties that have default values set at the time of creation of the entity. If needed, these values can also be changed on a per request basis by the client at the time of starting the observe relationship. These properties are the *Notifications Aggregation Window*, the *Entity Operation* and the *Entity Precision*. We will explain these properties in detail in Section 5.

When the client is no longer interested in observing the group, it can terminate the observe relationship with the entity resource in the same way it would terminate an observe relationship with any other observable CoAP resource. The EM in turn then stops observing all the group members in the same way and removes the client from the list of observers.

5. Optimizing the number of client notifications

In the previous section, we have described our solution that allows CoAP clients to observe a group of CoAP resources by issuing a single CoAP observe request to the entity resource at the EM. This approach not only makes it easy to start observing several resources, but also opens new possibilities to combine and manipulate notifications at the EM before sending them to the observer. This has the benefit that intelligence is brought closer to the data sources, which helps reduce network traffic and improve its responsiveness. Consider the simple topology shown in Figure 9 consisting of an LLN connected to the Internet via a gateway, which is also acting as the Entity Manager. At the EM, an entity has been defined, which is being observed by three different clients that have different capabilities and different types of Internet connection and that are observing the group for different purposes.

The Internet server has a fast Internet connection and needs to store all the changes that occur to all members of the group.

The smartphone is connected to the Internet via an expensive data package and needs to be notified only when certain changes occur to the group as such.

The smart object in the LLN is a constrained device containing an actuator that needs to be adjusted according to the status of the complete group.

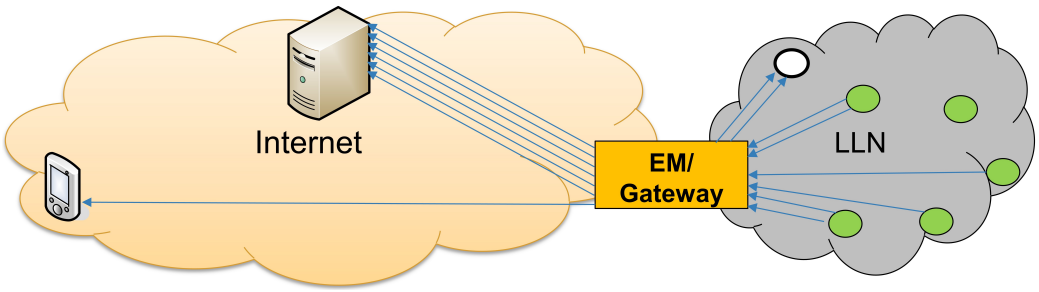


Figure 9: When an observed group member notifies the EM that its resource value has changed, the EM decides if it should notify observing clients about this change based on a set of per client configurable entity properties. This way, clients can elect to be notified about all changes of the individual group members, or just when certain changes occur to the group as such.

In the simplest way of observing an entity, the EM manager acts as a forwarder of notifications. That is, the number of client notifications sent to the client (client notifications) will be the sum of all the notifications that the EM has received from the group members (member notifications). This will satisfy the needs of the Internet server of knowing all the changes that occur to the individual members. However, this is clearly not what the smartphone and the smart object need to know and possibly could handle. In order to satisfy the needs of these observing clients, the EM should be able to offer ways that allow the clients to specify what types of notifications they are interested in receiving. In this section, we explore some techniques that we have added to the EM to achieve this goal. For each technique we derive a simple mathematical model that helps understand the benefits of using this technique. These techniques can be applied individually or combined in order to achieve even better optimization of the client notifications. In Section 6.3 we will validate these models experimentally.

5.1. Entity Operation

When a client observes a group, there are cases where the client is interested in observing all changes triggered by the group members. However, there are also certain cases where the client only needs to know about changes of the group status as such. For example, an environmental monitor might be interested in observing the maximum temperature of all sensors in a large cool storage room, in order to raise an alarm if this value exceeds a certain threshold. In order to be able to let the client observe the group according

to the client's needs, the EM allows clients to select between different Entity Operations when they register their entity observation requests. Clients can do so by adding the Uri-Query `eo={lst,avg,min,max, ...}` to the observe registration requests. The default Entity Operation `lst` notifies the observer with a list of values of all entity members each time the EM gets a member notification. The other Entity Operations perform arithmetical operations on the individual member values. For this the EM needs to be able to understand the data formats, so strict typing needed. At this moment this is the responsibility of the entity creator, but the EM architecture allows automating this during the validation of the entity and during its usage.

Different clients can observe the same entity using different Entity Operations and thus might get different notifications from the EM for the same change in the observed entity. Consider again the topology shown in Figure 9, where the cloud server might observe using `eo=lst` and will thus be notified about all changes in the group. On the other hand the smartphone is using `eo=max` and will thus be notified only if the maximum value of all members changes. The latter case might help reduce the cost of mobile communication and reducing the power consumption of the smartphone in processing not needed information.

In order to calculate the savings in the number of notifications as a result of using Entity Operations, let us assume that the client is observing an entity with g members and thus the EM is observing all of its g members. At each sampling point, every member might change its value with a probability p_{change}. If the value changes, the EM is notified, which in turn might notify the client, based on the selected Entity Operation. The number of member notifications that the EM receives n_m is the sum of all notifications sent by the individual members:

$$n_m = s \times \sum_{i=1}^{g} p_{change_i} \tag{1}$$

where s is the number of sampling points. For simplicity, we assume that all members have equal probability of value change and thus

$$n_m = s \times g \times p_{change} \tag{2}$$

In the case of using `eo=lst`, the EM sends a notification to the observing client each time it receives a notification from any member. In this case, the number of client notifications n_{lst} equals the number of received member

notifications n_m and the ratio r between client and member notifications is simply

$$r_{lst} = \frac{n_{lst}}{n_m} = 1 \tag{3}$$

In the case of using `eo=max`, the client is interested only in the maximum value of all observed members and thus the EM notifies the client only when the maximum value changes. This means that the EM sends notifications to the client in the following cases:

- The member with the current maximum value (probability: $1/g$) changes its value (probability: p_{change}).
- A member that does not currently have the maximum value (probability: $(g-1)/g$) changes its value (probability: p_{change}) and the new value is higher than the current maximum (probability: p_{higher})

On the other hand, the EM does not send notifications to the client when a member that does not currently have the maximum value changes its value and the new value is still lower than the current maximum. For the sake of simplicity, we neglected the cases in which more than one member have the exact maximum value. Accordingly, it is expected that the number of client notifications n_{max} is less than the number of received member notifications n_m and we can calculate n_{max} as follows:

$$\begin{aligned} n_{max} &= s \times g \times \left(\frac{1}{g} \times p_{change} + \frac{g-1}{g} \times p_{change} \times p_{higher} \right) \\ &= s \times p_{change} \times (1 + (g-1) \times p_{higher}) \end{aligned} \tag{4}$$

We can calculate the ratio between client and member notifications as follows:

$$r_{max} = \frac{n_{max}}{n_m} = \frac{1 + g \times p_{higher} - p_{higher}}{g} \tag{5}$$

This equation means that the ratio r_{max} between client and member notifications when using the `max` Entity Operation is independent from p_{change}. Moreover, r_{max} drops exponentially with the increase of the group size g and drops linearly with the drop of the probability that a new member notification's value is higher than the same members previous notification's value p_{higher}. These relationships are visualized in the 3D plot in Figure 10.

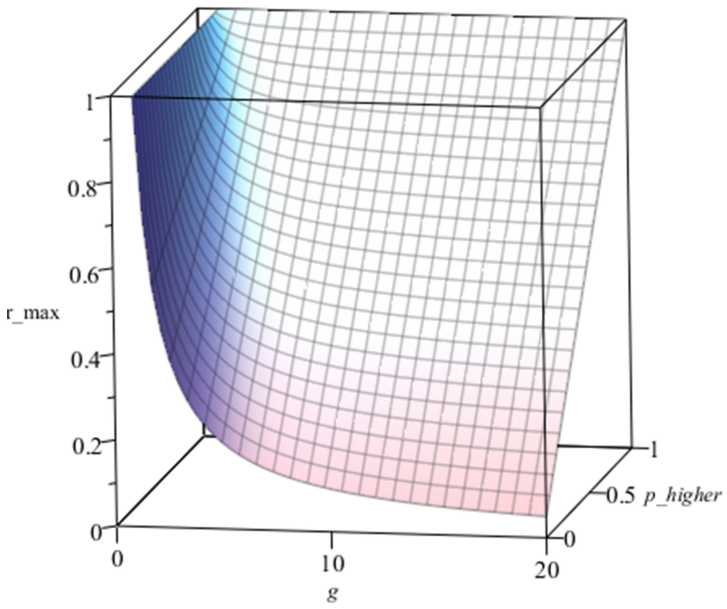


Figure 10: The ratio r_{max} between client and member notifications when using the `max` Entity Operation has a linear dependency on p_{higher} and a reverse exponential dependency on the group size g.

5.2. Entity Precision

When the EM applies the Entity Operations to aggregate the member notifications that contain sensor values into a single client notification, it uses the default *Entity Precision*, which depends on the Entity Operation itself. The Entity Precision specifies the number of digits after the comma for real numbers – i.e., a precision of two results in numbers with two decimal digits whereas a precision of zero results in integer values. For example, for the Entity Operations `lst`, `max` and `min`, the EM sends the client notifications using the same precision as the one used in the member notifications. For `eo=avg`, the EM uses a precision of six decimal digits for the calculated average value.

In some cases, the client might not be interested in the full precision of the measurements obtained by the individual sensors. As such, it can instruct the EM to send the results using a specific precision. When the client creates the entity or when it tries to use an existing entity, the client has the possibility to specify this precision for the values it receives from the EM by adding the Uri-Query `precision=x`, where `x` is the desired precision. If the client does not specify a precision, the EM passes the results to the client using the default precision.

To illustrate how reducing the precision can reduce the number of client notifications, please consider Figure 11. This figure is based on the temperature logs of Asheville regional airport for the first 12 hours of 2007[1]. It shows the notifications that would be generated as a result of the dew point temperature using two precisions: the original data precision (1 decimal point), and a reduced precision (integers only). In this particular example reducing the precision will result in 8 notifications instead of the original 16 notifications, i.e., 50% reduction in the number of notifications. Using a lower precision becomes even more relevant when applying Entity Operations that do not result in rounded numbers, e.g., when computing averages.

Consider again the topology shown in Figure 9. Whenever a client uses `eo=lst` with default precision, the EM sends a new notification to the observing client each time it receives a new notification from any member with the values received from the members in the same precision the EM received them. In this case, the number of sent notifications to the client n_{lst} equals the number of received notifications from the members n_m and the ratio

[1]Data source: http://cdo.ncdc.noaa.gov/qclcd/qclcdhrlyobs.htm

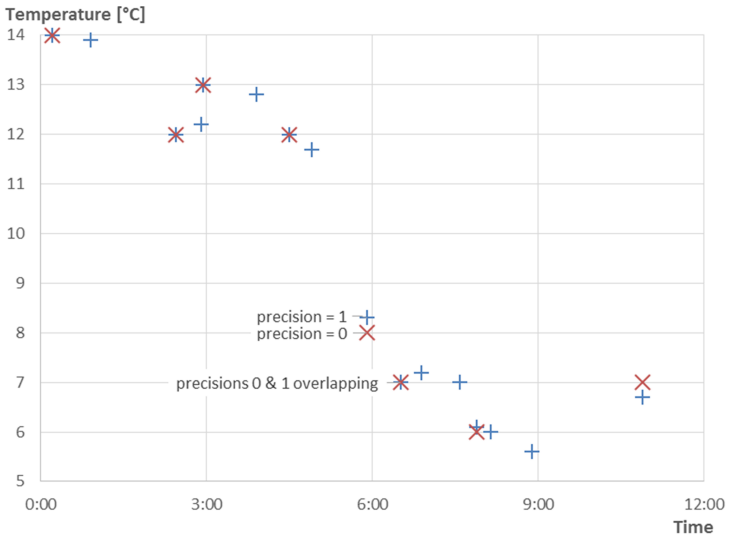


Figure 11: Reducing the precision of notifications can considerably reduce the number of client notifications.

between sent and received notifications equals one - as calculated in Equation (3).

When the client specifies a precision for the entity that is lower than the precision of the individual members, the EM first rounds the individual values to the specified precision and notifies the client only if these rounded values have changed. Thus, the number of client notifications is expected to be less than the number of member notifications. Of course, selecting a higher precision than the individual members does not have any effect on reducing the number of client notifications. The amount of reduction in the number of client notifications depends on p_{same} which is the probability that the changed value by any given member is still the same as the last value, after rounding both values to a lower precision. The lower the precision is, the higher the p_{same} will be. Since the EM combines different members notifications together, the ratio of client to member notifications for a lower precision $r_{precision}$ is independent of the group size and can be calculated as:

$$r_{precision} = 1 - p_{same} \tag{6}$$

5.3. Notifications Aggregation Window

The third technique available for clients to reduce the number of notifications that EM sends to them is the use of a *Notifications Aggregation Window*. This entity property is set as the Uri-Query `obswin=w`, where `w` is the number of seconds that the EM can buffer member notifications in order to allow other member notifications to arrive at the EM before combining them together and sending a single client notification. All notifications from one or more members that arrive during this period are sent in an aggregated manner in a single client notification, once the period expires. Of course using the Notifications Aggregation Window has the disadvantage that member notifications are delayed – in the worst case up to value of the Notifications Aggregation Window itself. However, often this delay can be tolerated for the sake of the reduction of the number of notifications.

When the EM is not using a Notifications Aggregation Window ($w = 0$), then the ratio between client and member notifications r equals one. Every member notification results in a client notification. As w increases r exponentially drops and approaches 0 as w approaches infinity. In that case, the EM will wait infinitely and will no longer send any client notifications (Figure 12). Of course, this is a simplification as we have ignored the fact that at a certain moment, the aggregated information might no longer fit into a single packet. In that case, the notification needs to be fragmented into multiple packets (e.g., CoAP blocks) or multiple notifications, each fitting into a single packet, need to be sent. For simplicity, this behavior is ignored.

If the member notifications are completely independent from each other, they will arrive randomly at the EM. Consequently, every member notification will be delayed for a time d that is between 0 and the Notifications Aggregation Window w and the average delay $\bar{d}$ over all member notifications equals its half, i.e.,

$$\bar{d} = w/2 \tag{7}$$

In certain cases, when an environmental change occurs, this change is registered by many sensors in a relatively short period. This means that the member notifications are not completely independent, but have a certain correlation. This correlation will likely affect the distribution of the member notifications over the Notifications Aggregation Window. If the distribution of the arrivals tends to be concentrated more at the beginning of the window, the member notifications will be delayed on average longer than $w/2$.

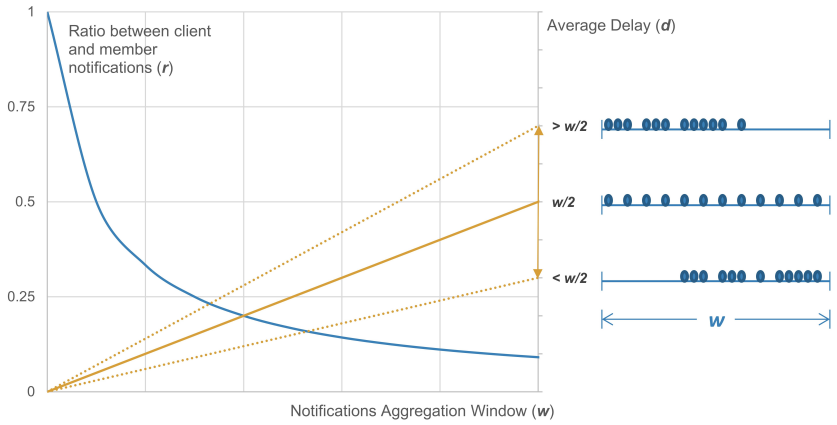


Figure 12: Ratio between client and member notifications and the average delay for member notifications as a function of the Notifications Aggregation Window.

Similarly, if the distribution of the arrivals is more concentrated at the end of the window, then the average delay will become less than $w/2$ (See right side of Figure 12). Also, in both cases the Notifications Aggregation Window is larger than needed and is thus suboptimal. The optimal window size would be a window that is just large enough to include all correlated member notifications and this window should start the moment the first member notification of a batch of correlated member notifications arrives. Clients that observe the entity might have knowledge about the optimal window size, for example from knowing the specs of the used sensors and the properties of the observed environment. Alternatively, the EM may automatically adapt the window size based on the history of the distribution of notification.

In order to visualize the effect of using the Notifications Aggregation Window, we consider an entity with five members. In Figure 13 we show the effect of the size of the Notifications Aggregation Window (w) on the number of client notifications (n) and the average introduced delay (d) for two sample arrival patterns of member notifications. The figure contains two sets of five correlated member notifications that arrive at the EM with different distributions. The empty circles show when member notifications arrive at the EM and the filled circles show when client notifications are sent by the EM to the observing client. When $w = 0$, the EM notifies the observing client as soon as it receives a notification from the observed members. As $w = 0$ increases, n generally decreases to reach 1 when w is large enough to aggregate all member notifications into a single client notification. i.e., $w =$

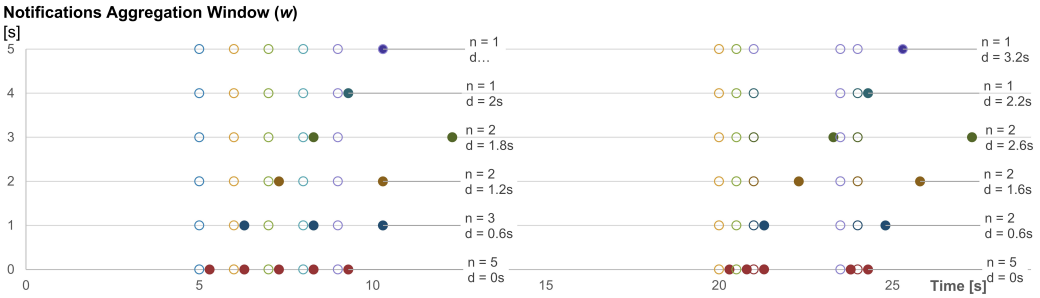


Figure 13: Client notifications generated by the EM as a result of receiving member notifications. The number of client notifications (n) and the average delay (d) depend on the number of notifications received, the Notifications Aggregation Window size (w) and on the distribution of the received notifications. Note: the empty circles indicate when member notifications are received by the EM; the filled circles indicate when client notifications are sent by the EM to the observer.

$4s$ in both examples. Increasing w beyond 4 cannot optimize n any further; it just makes d worse. Although in the above examples the member notifications were generated by different members, in fact the number of notifications sent from the EM to the client does not change based on the source of the notifications. If a certain member sends more than one notification in the same Notifications Aggregation Window, the newest value will replace the older values. This means that the client might skip a change of that member if the change occurred within a period shorter than w.

6. Implementation and Evaluation

Our solution described above enables the use of unicast messages as an alternative to using multicasts for realizing CoAP group communication. By relying only on standard CoAP unicast messages, our solution is able to extend the CoAP observe option to be suitable for group communication as well. In order to evaluate our solution and to show how it can be used in a real-world scenario we have implemented it and built a demo box for demonstration purposes [28]. In this section we present the implementation of our solution followed by functional and performance evaluations.

6.1. Implementation

The key in our group communication solution is the Entity Manager. We have implemented the Entity Manager functionality on the gateway of

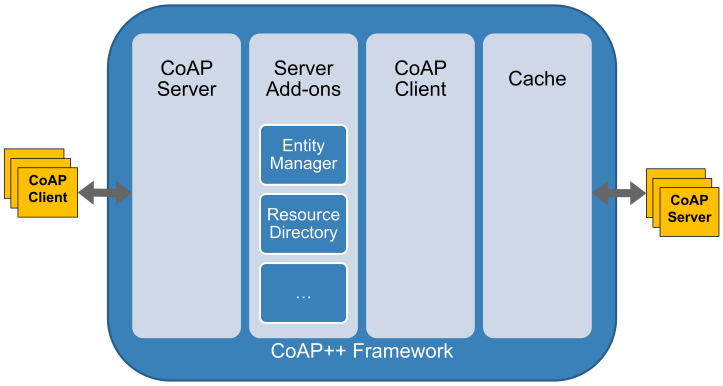


Figure 14: Location of the Entity Manager within the CoAP++ Architecture

the LLN using the CoAP++ framework [29]. The CoAP++ framework and the Entity Manager implementation on top of it have been realized in *Click Router*, a C++ based modular framework that can be used to realize any network packet processing functionality [30]. The CoAP++ framework is modular and can be used for various scenarios (Figure 14). Our EM needs to act as a CoAP client and as a CoAP server at the same time. The CoAP client functionality is needed to communicate with the entity members while the CoAP server functionality is needed to serve the clients of the entities. To realize those client and server functionalities, we use the *CoAP Client* and *CoAP Server* modules from the CoAP++ framework. The CoAP Server module functionality can be extended by developing add-ons to be plugged in to it. One such an add-on is the *Resource Directory*, which extends the framework with a CoAP Resource Directory (see Section **??**). Our EM is also implemented as an add-on to the CoAP Server module. Furthermore, since the CoAP++ framework already contains a *cache* module, we do not implement this functionality in the EM itself, but use the framework's cache module when it is needed. Finally, the CoAP++ framework includes a *Dummy Sensor* module (not shown in Figure 14 for simplicity). This module allows developers to define one or more software sensors in the framework. These software sensors offer several CoAP resources that can be queried and observed by CoAP clients. This way it becomes possible to perform initial testing against CoAP servers without the need for real sensors.

In addition to the software sensors from the Dummy Sensor module we have used Zolertia Z1's [31] and Rmoni RM090 [32] boards as members. On

these boards we have run a development version[2] of the popular Contiki operating system [33]. This version of Contiki was the most recent development version when we started our experiments. It supports multicast and includes a stable implementation of CoAP, namely the Erbium CoAP server [34]. In addition to providing base CoAP functionality, Erbium also supports the observe extension. Our group communication approach does not require any changes on the CoAP enabled constrained devices. However, in order to demonstrate how the EM can use resource profiles to validate entities, we have added resource profiles to the constrained CoAP servers.

6.2. Functional Evaluation

The functionality for creating, validating, using and deleting observable entities has been implemented as described above. In this subsection we demonstrate the main functionalities of the implementation using a series of screen-shots covering the life cycle of an observable entity (Figure 15). These screen-shots are taken using the CoAP++ client Graphical User Interface (GUI).

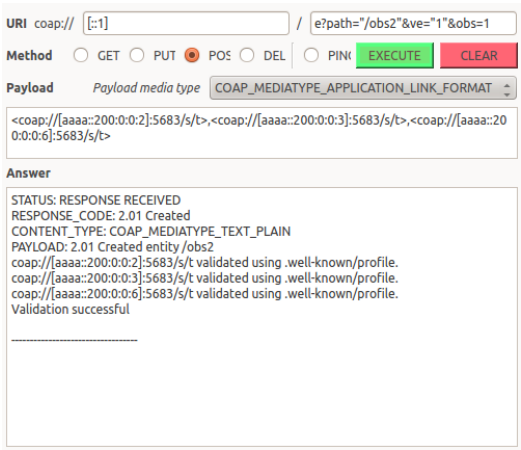
In Figure 15a the client requests to create an observable entity of three members. The members are three temperature resources on RM090 sensors. The EM validated the members by querying the profiles of theses resources and validated that all the resources are observable by checking /.well-known/core of all sensors. The entity was created successfully and is ready for use since it passed the validation.

Next, the client checks the profile of the newly created entity (Figure 15b). The profile shows that the entity supports four Entity Operations: `lst, avg, min` and `max` whereas `lst` is the default operation since it is the first operation in the list of supported operations.

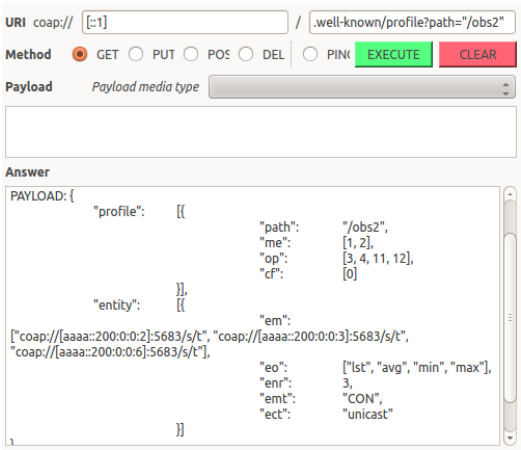
Next, the client starts observing the newly created entity without specifying any Uri-Queries and thus the default entity properties (Entity Operation, Entity Precision and Notifications Aggregation Window) are used. Figure 15c shows how the client is getting a notification containing all members values each time any member changes its value.

Since the client is actually interested in the average temperature of all three sensors, the client cancels the current observe relationship and starts observing the entity using the Entity Operation `avg` (Figure 15d). By doing

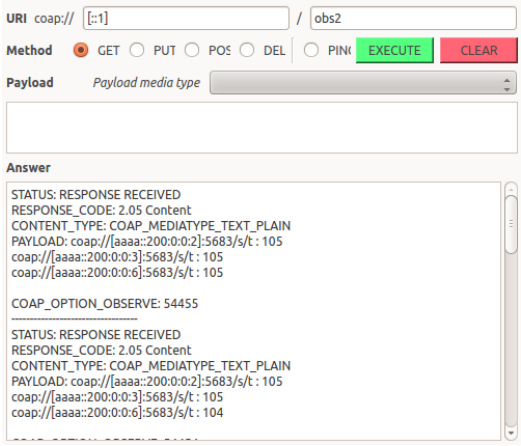
[2]snapshot from the Contiki master branch on github on 2 July 2014



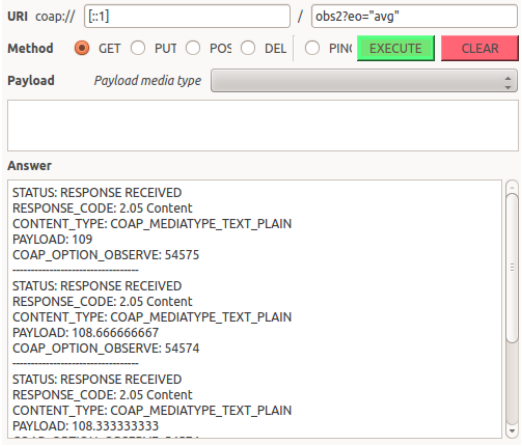
(a) Creating observable entity obs2



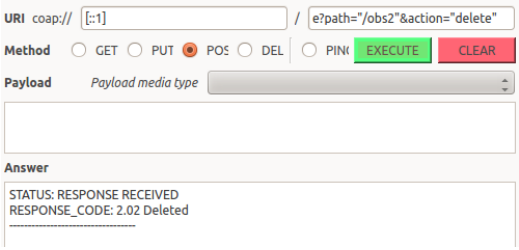
(b) Profile of obs2



(c) Observing with default properties



(d) Observing with entity operation avg



(e) Deleting the entity obs2

Figure 15: Screen-shots using the CoAP++ client GUI to create, observe and delete an observable entity of three members.

so, the EM now calculates the average and only sends the calculated value to the client. If the average value does not change as a result of a change of one of the member values, no notification will be sent by the EM.

To complete the life cycle of an entity we show how the client deletes it in Figure 15e.

6.3. Performance Evaluation

The three communication participants of our approach are the clients, the EM and the entity members. Clients might be constrained devices (e.g., a light switch triggering a group of lights) or not constrained (e.g., a building management computer). In any case, the use of our approach generally requires less processing and memory on the client side since complexity is outsourced to the EM (Fog idea). Clients need to worry about only one observe relationship with the EM, instead of maintaining observe relationships with all group members. Furthermore, clients need to store only final processed results, instead of storing intermediate results and processing them later on.

The EM implementation is highly dependent on the type of device on which EM is running. In our case, we used Linux, powered with click modular router and the CoAP++ Framework.

Our solution requires no additions to the CoAP implementation of the constrained devices acting as group members. Thus, it has no overhead in terms of memory and CPU. Since the EM behaves as a proxy between the clients and the constrained devices it also limits the number of observe relationships that constrained CoAP servers have to maintain. This not only helps to reduce the memory that should be reserved for maintaining the observe relationships, but also results in fewer packets sent by the constrained devices. This reduces energy consumption by the constrained devices, since a major source for it is radio transmissions, which is directly related to the number of notifications. The exact values for energy consumptions are hardware and MAC protocol dependent. Thus, in this paper we focus our evaluation on the number of notifications. For a discussion of the relation between the number of notifications and energy consumption, we refer to [35].

In Section 5 we have presented three techniques that clients, which want to observe a group, can use individually or combined in order to optimize the number of notifications they receive from the EM. For each technique we provided a simple mathematical model for the reduction in the number of client notifications. In this section we present the results of a series of

Contiki settings	Version	Snapshot from the Contiki master branch on github on 2 July 2014
	Radio Duty Cycling (RDC)	ContikiMAC
	Media Access Control (MAC)	CDMA
	Routing Protocol	RPL
Cooja settings	Radio Medium	Unit Disk Graph Medium (UDGM): Distance Loss
	Transmit Ratio	100 %
	Receive Ratio	100 %

Table 1: Contiki and Cooja network simulator settings

experiments that we have conducted in order to validate these models. In those experiments we have used as entity members either software sensors from the CoAP++ Dummy Sensor module or RM090 sensors simulated in Cooja (the Contiki network simulator). In Table 1 we summarize the most important contiki and Cooja network simulator settings.

6.3.1. Entity Operation

As described in Section 5.1 the Entity Operation (eo) property allows the client to select an operation that is applied on the values in the notifications from the entity members before sending the result the client. When using `eo=lst`, it is expected that the number of client notifications equals the number of member notifications regardless of the group size. For the other operations it is expected that the ratio between client and member notifications drops exponentially as the group size increases. Please note that the exact speed at which the ratio drops depends on the selected entity operation and on the pattern of changes that occur to the individual members as can be seen in Equation (5) and Figure 10. This behavior can indeed be observed in Figure 16 for the three tested operations `lst`, `avg` and `max`.

Each dot in Figure 16 represent an experiment while the continuous lines represent the average values. In these experiments we used software sensors as members. This has the benefit that it allows us to control the pattern in which the values of the sensors change over time and consequently the pattern of member notifications. In the experiments all sensors started from the same value. Every 5 seconds each sensor randomly decides if it wants to

Figure 16: The ratio between the number of client notifications to the number of member notifications for different Entity Operations (lst: list all values; max: maximum value; avg: average value)

keep its current value, slightly increase it or slightly decrease it. This way we achieve that the values that we want to observe are following a continuous function but with random changes. By using the same algorithm for deciding whether there is a change (and thus a notification) at the members, we make sure that value of p_{change} in Equation (5) is constant across experiments. The experiments had different durations and the number of member notifications varied between them. The average number of notifications sent per member per experiment was 35, the minimum was 8 and the maximum was 103 notifications.

6.3.2. Entity Precision

The second technique clients can use in order to optimize the number of notifications they receive from the EM is the Entity Precision (Section 5.2). If a client is not interested in the full precision of the sensor data, it can request the EM to reduce the precision. Here again, reducing the precision also typically reduces the number of client notifications, since consequent member notifications values might be the same when their precision is reduced, not resulting in a new client notification.

As calculated in Equation (6), the ratio $r_{precision}$ between the number of client notifications to member notifications is independent from the group size. In fact $r_{precision}$ depends only on p_{same} which is the probability that two

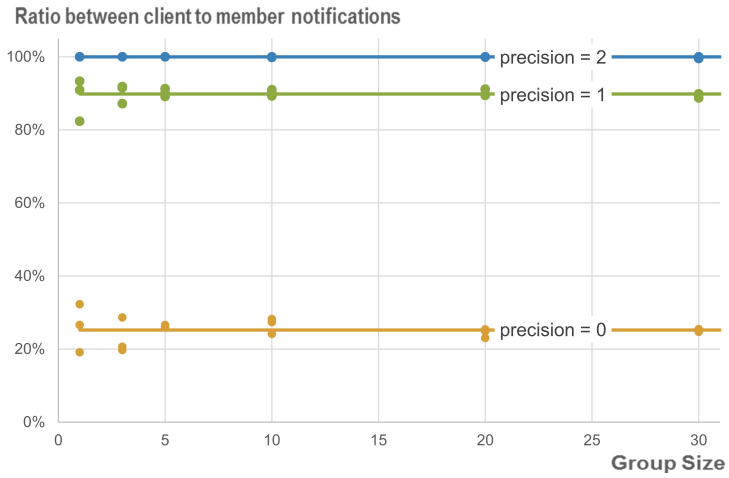


Figure 17: Reducing the precision of the entity can reduce the number of client notifications. The reduction ratio is independent from the group size.

consequent values from a single member will be the same if their precision is reduced. We have validated this behavior experimentally and show the results in Figure 17. In these experiments we used the Entity Operation `lst` which has no effect on $r_{precision}$ in order to be able to focus on the effect of reducing the precision on $r_{precision}$.

The Figure shows the results of three sets of experiments for three different precisions: two, one and zero. As members, we have used software sensors in order to be able to change the precision of the resources they represent and have the precision set to two digits. Thus, when the EM used the same precision with the observing clients, all notifications were forwarded and hence $r_{precision} = 100\%$. When the client requested the reduction of the precision by one digit (i.e., $precision = 1$) and by two digits ($precision = 0$) $r_{precision}$ was reduced to 89.8% and 25.2% respectively. The exact values of $r_{precision}$ depend on the pattern on which the data changes. This is second reason why we used the software sensors and the same function for changing their values across all experiments (similar to Section 6.3.1). For very small group sizes (three members or fewer) the deviation between the results of different experiments is noticeable in the graph. However, as the groups get larger, the deviation gets smaller since the total number of received notifications from all members gets larger resulting in better accuracy of the experiments. The experiments had different durations and the number of notifications sent by the members varied between them. The average number of notifications sent

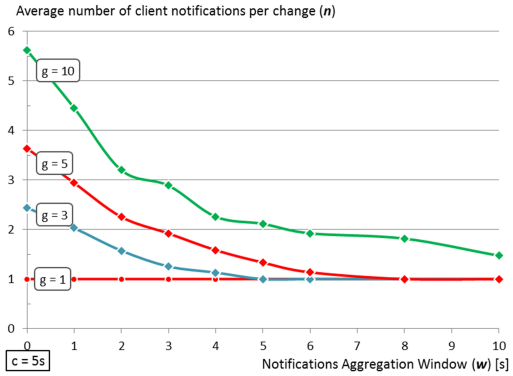
per member per experiment was 81, the minimum was 9 and the maximum was 275 notifications.

6.3.3. Notifications Aggregation Window

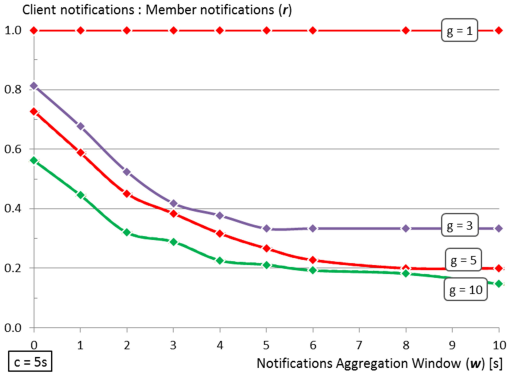
In addition to the Entity Operation and Entity Precision that are explained in the previous subsections, clients can specify a Notifications Aggregation Window w (see Section 5.3). This window tells the EM the number of seconds up to which the EM can delay notifications before sending them to the client. By allowing the EM to delay some notifications, it becomes possible for the EM to combine several member notifications into a single client notification and thus decreasing the number of notifications the client receives. The reduction in the number of client notifications depends on the timing of the member notifications. If multiple sensors monitor a single environment, then it is likely that a change in this environment will be registered by those sensors in a relatively short period. If these sensors are the members of an observed entity, then the notifications that the individual members send to the EM are not independent from each other; they are correlated. When using an optimal Notifications Aggregation Window, the EM will capture all of those member notifications into a single client notification.

In order to investigate the effect of introducing the Notifications Aggregation Window on the number of client notifications, we have conducted several experiments using members running in Cooja network simulator. The use of Cooja allows us to accurately control the timing of notifications sent by the individual members to the EM so that we are able to create tests in which the notifications from different members arrive at the EM within a certain period. We call this period the *change window* c. In all experiments in this section we introduce an environmental change that can be monitored by all members. Each member notifies the EM about the change after a random time between zero and c. This way we simulate the correlation between the notifications from different members. We have conducted several experiments to explore the effect of the three variables: Notifications Aggregation Window (w), change window (c) and group size (g) on the number of client notifications (n) and the ratio between client and member notifications (r). In each set of experiments we kept two variables constant and changed the third variable and measured its effect. We let each experiment run for a duration long enough to allow at least 50 notifications from each member.

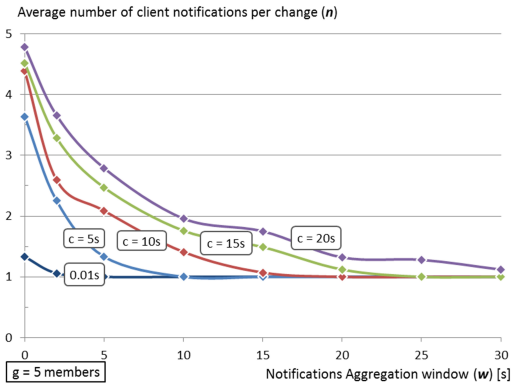
Effects of the size of Notifications Aggregation Window. In the first



(a) n vs w; $c = 5s$



(b) r vs w; $c = 5s$



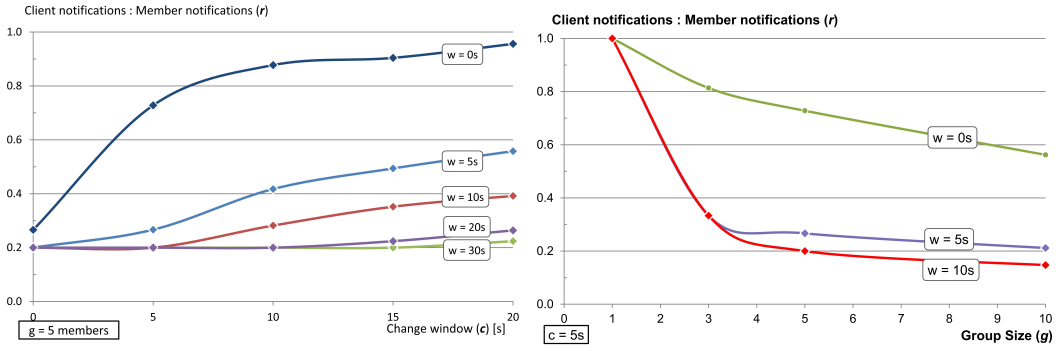
(c) n vs w; $g = 5$

Figure 18: The effect of the Notifications Aggregation Window size (w) on the number of client notifications (n) and the ratio between client and member notifications (r). (c: change window, g: group size.)

set of experiments (Figure 18a) we used a fixed change window ($c = 5s$) and changed the Notifications Aggregation Window ($w = 0 \rightarrow 10s$) and measured the average number of client notifications (n) for different group sizes ($g = 1, 3, 5, 10$). As expected, the graph shows how an increase in w results in an exponential decrease in n, as multiple member notifications are aggregated into a single client notification than smaller groups. Nevertheless, when w becomes sufficiently large, a single notification is sent to the client, containing all member notifications. Obviously, larger groups result in more notifications than smaller groups. However, if we normalize the number of notifications by the group size, we get the ratio between client and member notifications r regardless of the group size. Figure 18b shows how larger groups achieve a smaller r (better reduction of the number of client notifications). The reason for this is that with bigger groups the number of client notifications that arrives at the EM in a single w gets larger. The functions converge to $1/g$ which means that when w is big enough the EM combines all correlated notifications from the individual members in one client notification.

In the second set of experiments (Figure 18c) we used a fixed group size ($g = 5$) and changed the Notifications Aggregation Window ($w = 0 \rightarrow 10s$) and measured the average number of client notifications (n) for different change window sizes ($c = 0.01s, 5s, 10s, 15s, 20s$). Here we see that larger change windows result in more client notifications being sent. This is due to the fact that with the increase of c it becomes less likely that member notifications will fit in the same Notifications Aggregation Window. When w becomes sufficiently large, the EM sends a single notification to the client.

Effects of the change window size. In the third set of experiments (Figure 19a) we used a fixed group size ($g = 5$) and changed the change window ($c = 0.01 \rightarrow 20s$) and measured the ratio between client and member notifications (r) for different Notifications Aggregation Window sizes ($w = 0s, 5s, 10s, 20s, 30s$). One can clearly see here that when c is very small, all member notifications are aggregated in one client notification and thus $r = 1/g = 0.2$ since $g = 5$. Please note that c can not get a lot smaller than 0.01s since this is close the frame transmission time. If two members try to transmit with a very high degree of synchronization, collision between them becomes very likely



(a) n vs c; $g = 5$ (b) n vs g; $c = 5s$

Figure 19: The effects of the change window (c) and the group size (g) on the ratio between client and member notifications (r) for various Notifications Aggregation Window sizes (w).

and causes the MAC protocol to back-off before trying to retransmit later.

Effects of the group size. In the fourth set of experiments (Figure 19b) we used a fixed change window size ($c = 5s$) and changed the group size ($g = 1 \rightarrow 10$ *members*) and measured the ratio between client and member notifications (r) for different Notifications Aggregation Window sizes ($w = 0s, 5s, 10s$). Here again the reverse exponential relationship between r and g is clearly visible. An interesting aspect that this figure reviles is that even when the $w = 0$ the EM manges to reduce client notifications when they are correlated as in these experiments. This is implementation dependent, as our implementation periodically checks if the EM has pending client notifications to send. When the member notifications are correlated, sometimes more than one notification arrive within this short period. When the group gets bigger, the chance that this happens get bigger as well.

6.3.4. Combination of properties

The previous subsections have shown that the EM provides three techniques that allow the client to reduce the number of notifications they get. Clients can use any of those techniques individually or may elect to combine any two or all three of them to achieve an even better overall optimization. To demonstrate this please consider Figure 20, where the results of four set

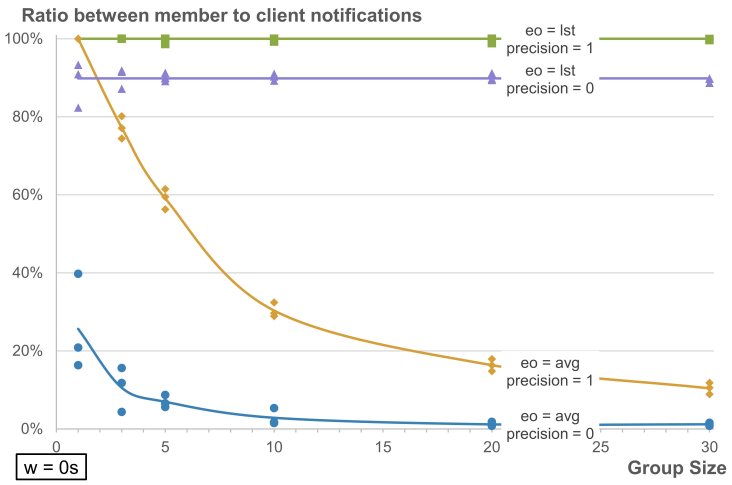


Figure 20: Combining entity properties achieves even better optimization.

of experiments are shown. First the client does not select any optimization technique and thus receives all member notifications as individual client notifications (i.e., $r = 1$). In the second experiment the client chooses to reduce the precision from 1 decimal digits to integer values and gets a reduction of about 10% of the number of the notifications regardless of the group size. In the third experiment the client uses the Entity Operation `avg` to get notified when the average value (with 1 digit precision) of all members changes. Finally, the client selects to combine both techniques (`eo=avg` and `precision=0`) and gets the benefits of both techniques combined.

7. Conclusions and Outlook

In this paper we have tackled the problem of observing a group of CoAP resources by further exploring the potential of our unicast-based CoAP group communication approach. We have demonstrated that by relying on standard CoAP unicast messages, it becomes possible to observe CoAP groups effectively. We have also shown that several techniques can be applied in order to further reduce the number of notifications sent to the observer. The amount of reduction in the number of notifications depends on the size of the group, the entity operation, and on the correlation between the values of different members. In our test examples we were able to reduce the number of notifications down to just 1% of the original number by reducing the data precision and using the Entity Operation `avg`. This approach fits within the

current spirit of distributed processing or fog computing, where processing functionality is moved closer to the data sources, as bandwidth towards the Cloud is not infinite, might be costly or might cause to high latencies.

An advantage of our approach is that it requires no changes to sensors, i.e., one can add it afterwards to any deployed CoAP network. Also it requires no changes to the clients, since groups behave as any other CoAP source. In the future, we plan to evaluate our approach in more detail. We will test the principles by using sensor data collected in real-life settings, both uncorrelated and correlated data.

An open challenge in our approach is that machines need to know more details about the data they are getting from the various sources in order to be able to process this data correctly. This includes the need for strict typing of data to be able to understand the contents of the resources. In addition to the CoRE link format, several initiatives that provide guidelines and standards in this regard are noteworthy. The *Internet Protocol for Smart Object (IPSO) Alliance* has published an *Application Framework* that recommends a classification of resources based on their functionality by defining a set of resource types [13]. Another example is the *Open Mobile Alliance* (OMA) *Lightweight M2M* (LWM2M), which is an open industry standard that specifies a way to remotely manage a wide range of M2M devices and connected appliances in the IoT [14]. In the future, we will build upon such standards to further extend the CoAP++ framework to automatically interpret resource representations (content formats), data units, possible operations, etc. More work is needed to support automatic conversion between content formats and data units. Also standardization is needed to some extent, e.g., profiles of groups.

We like to further compare our approach with MQTT and see whether we can also apply QoS. In order to improve the performance of our solutions, we like to let the EM automatically adapt the Notifications Aggregation Window size based on the history of the distribution of notifications. Finally, further improvements might be possible by intervening at the routing and MAC levels in the LLN.

Acknowledgments

The research leading to these results has received funding from the European Union's Seventh Framework Programme (FP7/2007-2013) under grant

agreement no 258885 (SPITFIRE project), from the iMinds ICON projects O'CareCloudS and a VLIR PhD scholarship to Isam Ishaq.

[1] I. Ishaq, D. Carels, G. K. Teklemariam, J. Hoebeke, F. V. d. Abeele, E. D. Poorter, I. Moerman, P. Demeester, Ietf standardization in the field of the internet of things (iot): A survey, Journal of Sensor and Actuator Networks 2 (2) (2013) 235–287.

[2] Constrained restful environments (core), [Online; accessed 19 March 2014] (2014).
URL http://datatracker.ietf.org/wg/core/

[3] Z. Shelby, K. Hartke, C. Bormann, The constrained application protocol (coap), RFC 7252, IETF (June 2014).
URL http://tools.ietf.org/html/rfc7252

[4] A. Rahman, E. Dijk, Group Communication for the Constrained Application Protocol (CoAP), RFC 7390, IETF (Oct. 2014).
URL http://www.ietf.org/rfc/rfc7390.txt

[5] I. Ishaq, J. Hoebeke, F. Van den Abeele, J. Rossey, I. Moerman, P. Demeester, Flexible unicast-based group communication for coap-enabled devices, Sensors 14 (6) (2014) 9833–9877.

[6] I. Ishaq, J. Hoebeke, V. den Abeele, Coap entities, Internet-Draft draft-ishaq-core-entities-00, IETF (June 2013).
URL http://tools.ietf.org/html/draft-ishaq-core-entities

[7] K. Hartke, Observing resources in coap, Internet-Draft draft-ietf-core-observe-16, IETF (December 2014).
URL http://tools.ietf.org/html/draft-ietf-core-observe

[8] Cisco fog computing with iox, [Online; accessed 5 March 2015] (2015).
URL http://www.cisco.com/web/solutions/trends/iot/cisco-fog-computing-with-iox.pdf

[9] F. Van den Abeele, J. Hoebeke, G. K. Teklemariam, I. Moerman, P. Demeester, Sensor function virtualization to support distributed intelligence in the internet of things, Wireless Personal Communications (2015) 1–22.

[10] M. Jung, W. Kastner, Efficient group communication based on web services for reliable control in wireless automation, in: Industrial Electronics Society, IECON 2013-39th Annual Conference of the IEEE, IEEE, 2013, pp. 5716–5722.

[11] M. Antonini, S. Cirani, G. Ferrari, P. Medagliani, M. Picone, L. Veltri, Lightweight multicast forwarding for service discovery in low-power iot networks, in: Software, Telecommunications and Computer Networks (SoftCOM), 2014 22nd International Conference on, IEEE, 2014, pp. 133–138.

[12] Z. Shelby, Constrained restful environments (core) link format, RFC 6690, IETF (August 2012).
URL http://www.ietf.org/rfc/rfc6690.txt

[13] The ipso application framework, [Online; accessed 5 March 2015] (2012).
URL http://www.ipso-alliance.org/wp-content/media/draft-ipso-app-framework-04.pdf

[14] Oma lightweight m2m (lwm2m) protocol, [Online; accessed 5 March 2015] (2012).
URL http://openmobilealliance.hs-sites.com/lightweight-m2m-specification-from-oma

[15] F. Bonomi, R. Milito, J. Zhu, S. Addepalli, Fog computing and its role in the internet of things, in: Proceedings of the first edition of the MCC workshop on Mobile cloud computing, ACM, 2012, pp. 13–16.

[16] H. Hasemann, O. Kleine, A. Kröller, M. Leggieri, D. Pfisterer, Annotating real-world objects using semantic entities, in: Wireless Sensor Networks, Springer, 2013, pp. 67–82.

[17] H. Hasemann, A. Kroeller, M. Page, , [Online; accessed 6 June 2015] (2012).
URL http://iot-conference.org/iot2012/uploadfiles/file/ppt/F1B-1%20RDF%20Provisioning%20for%20the%20Internet%20of%20Things.pdf

[18] C.-D. Hou, D. Li, J.-F. Qiu, H.-L. Shi, L. Cui, Seahttp: A resource-oriented protocol to extend rest style for web of things, Journal of Computer Science and Technology 29 (2) (2014) 205–215.

[19] G. Bovet, G. Briard, J. Hennebert, A scalable cloud storage for sensor networks, in: 5th International Workshop on the Web of Things, 2014.

[20] Message queue telemetry transport mqtt, [Online; accessed 2 March 2015] (2015).
URL http://mqtt.org/

[21] W. Colitti, K. Steenhaut, N. De Caro, Integrating wireless sensor networks with the web, Extending the Internet to Low power and Lossy Networks (IP+ SN 2011).

[22] A. Dunkels, et al., Efficient application integration in ip-based sensor networks, in: Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Efficiency in Buildings, ACM, 2009, pp. 43–48.

[23] M. Nottingham, Web linking, RFC 5988, IETF (October 2010).
URL http://www.ietf.org/rfc/rfc5988.txt

[24] S. Loreto, P. Saint-Andre, S. Salsano, G. Wilkins, Known issues and best practices for the use of long polling and streaming in bidirectional http, RFC 6202, IETF (April 2011).
URL http://www.ietf.org/rfc/rfc6202.txt

[25] M. Buettner, G. V. Yee, E. Anderson, R. Han, X-mac: a short preamble mac protocol for duty-cycled wireless sensor networks, in: Proceedings of the 4th international conference on Embedded networked sensor systems, ACM, 2006, pp. 307–320.

[26] Z. Shelby, C. Bormann, Core resource directory, Internet-Draft draft-ietf-core-resource-directory-02, IETF Secretariat (November 2014).
URL http://tools.ietf.org/html/draft-ietf-core-resource-directory

[27] C. Jennings, Z. Shelby, J. Arkko, Media types for sensor markup language (senml), Internet-Draft draft-jennings-senml-10, IETF (October 2012).
URL http://tools.ietf.org/html/jennings-senml

[28] F. Van den Abeele, J. Hoebeke, I. Ishaq, G. K. Teklemariam, J. Rossey, I. Moerman, P. Demeester, Building embedded applications via rest services for the internet of things, in: Proceedings of the 11th ACM Conference on Embedded Networked Sensor Systems, ACM, 2013, p. 82.

[29] I. Ishaq, J. Hoebeke, J. Rossey, E. De Poorter, I. Moerman, P. Demeester, Facilitating sensor deployment, discovery and resource access using embedded web services, in: Innovative Mobile and Internet Services in Ubiquitous Computing (IMIS), 2012 Sixth International Conference on, IEEE, 2012, pp. 717–724.

[30] E. Kohler, R. Morris, B. Chen, J. Jannotti, M. F. Kaashoek, The click modular router, ACM Transactions on Computer Systems (TOCS) 18 (3) (2000) 263–297.

[31] Zolertia z1 platform, [Online; accessed 27 February 2015] (2015).
URL `http://www.zolertia.com/ti`

[32] Rm090 — rmoni wireless nv, [Online; accessed 27 February 2015] (2015).
URL `http://www.rmoni.com/en/products/hardware/rm090`

[33] Contiki: The open source operating system for the internet of things, [Online; accessed 27 February 2015] (2015).
URL `http://www.contiki-os.org/`

[34] Erbium (er) rest engine and coap implementation for contiki, [Online; accessed 27 February 2015] (2015).
URL `http://people.inf.ethz.ch/mkovatsc/erbium.php`

[35] G. K. Teklemariam, J. Hoebeke, I. Moerman, P. Demeester, Facilitating the creation of iot applications through conditional observations in coap, EURASIP Journal on Wireless Communications and Networking 2013 (1) (2013) 1–19.